Copyright

by

Allen Grogan Clement

2010

The Dissertation Committee for Allen Grogan Clement
certifies that this is the approved version of the following dissertation:

UpRight Fault Tolerance

Committee:

____
Lorenzo Alvisi, Co-Supervisor

____
Mike Dahlin, Co-Supervisor

____
Peter Druschel

____
Michael Walfish

____
Emmett Witchel

UpRight Fault Tolerance

by

Allen Grogan Clement, A.B.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2010

Acknowledgments

As much as this document is "mine," it would not exist without the support, assistance, and guidance of numerous people.

My advisors, Lorenzo Alvisi and Mike Dahlin, have been instrumental in the process of completing this document. Their guidance over the last several years has been invaluable and I would not be the person or researcher that I am today without them.

The other members of the thesis committee (Peter Druschel, Michael Walfish, and Emmett Witchel) have exhibited great patience and understanding during the writing process. Their insights and comments on the work have been greatly appreciated and improved the quality of this document.

Fellow graduate students make grad school both possible and bearable. Over the past 8 years I've had the pleasure of working closely with some great students in the LASR group: especially Amit Aiyer, Manos Kapritsos, Rama Kotla, Sangmin Lee, Harry Li, Prince Mahajan, Mirco Marchetti, J.P. Martin, Jeff Napper, Don Porter, Taylor Riche, Eric Rozner, Chris Rossbach, Srinath Setty, Yang Wang, and Ed Wong. Thank you all for your friendship, advice, help, beers, and patience—I would not have finished without you.

Sara Strandtman, the LASR administrator, was very important in getting me out the door. While I may have survived graduate school without her, life was much less stressful knowing that she was there to protect me from the bureaucracy.

I would like to thank my parents and sisters for putting up with me for all these years. Without you I wouldn't be here today.

Finally, I thank Nathalie for her patience and understanding. While this process has been trying for me, it has probably been more trying for her. I wouldn't be writing this today without her patience and for that I am eternally grateful.

Allen Grogan Clement

The University of Texas at Austin
December 2010

UpRight Fault Tolerance

Publication No. ____________________

Allen Grogan Clement, Ph.D.
The University of Texas at Austin, 2010

Co-Supervisor: Lorenzo Alvisi
Co-Supervisor: Mike Dahlin

Experiences with computer systems indicate an inconvenient truth: computers fail and they fail in interesting ways. Although using redundancy to protect against fail-stop failures is common practice, non-fail-stop computer and network failures occur for a variety of reasons including power outage, disk or memory corruption, NIC malfunction, user error, operating system and application bugs or misconfiguration, and many others. The impact of these failures can be dramatic, ranging from service unavailability to stranding airplane passengers on the runway to companies closing.

While high-stakes embedded systems have embraced Byzantine fault tolerant techniques, general purpose computing continues to rely on techniques that are fundamentally crash tolerant. In a general purpose environment, the current best

practices response to non-fail-stop failures can charitably be described as pragmatic: identify a root cause and add checksums to prevent that error from happening again in the future. Pragmatic responses have proven effective for patching holes and protecting against faults once they have occurred; unfortunately the initial damage has already been done, and it is difficult to say if the patches made to address previous faults will protect against future failures.

We posit that an end-to-end solution based on Byzantine fault tolerant (BFT) state machine replication is an efficient and deployable alternative to current *ad hoc* approaches favored in general purpose computing. The replicated state machine approach ensures that multiple copies of the same deterministic application execute requests in the same order and provides end-to-end assurance that independent transient failures will not lead to unavailability or incorrect responses. An efficient and effective end-to-end solution covers faults that have already been observed as well as failures that have not yet occurred, and it provides structural confidence that developers won't have to track down yet another failure caused by some unpredicted memory, disk, or network behavior.

While the promise of end-to-end failure protection is intriguing, significant technical and practical challenges currently prevent adoption in general purpose computing environments. On the technical side, it is important that end-to-end solutions maintain the performance characteristics of deployed systems: if end-to-end solutions dramatically increase computing requirements, dramatically reduce throughput, or dramatically increase latency during normal operation then end-to-end techniques are a non-starter. On the practical side, it is important that end-to-end approaches be both comprehensible and easy to incorporate: if the cost of end-to-end solutions is rewriting an application or trusting intricate and arcane protocols, then end-to-end solutions will not be adopted.

In this thesis we show that BFT state machine replication can and be used in

deployed systems. Reaching this goal requires us to address both the technical and practical challenges previously mentioned. We revisiting disparate research results from the last decade and tweak, refine, and revise the core ideas to fit together into a coherent whole. Addressing the practical concerns requires us to simplify the process of incorporating BFT techniques into legacy applications.

Contents

Acknowledgments **iv**

Abstract **vi**

List of Tables **xiii**

List of Figures **xv**

Chapter 1 Introduction **1**

Chapter 2 Failure models and fault tolerance **5**

2.1 Classifying node and network behaviors 6

2.1.1 Faulty behaviors 6

2.1.2 Correct behaviors 6

2.1.3 Cryptographic assumptions and notation 8

2.1.4 Network Behaviors 9

2.2 Fault tolerance 9

2.3 Why UpRight? 13

Chapter 3 Robust Performance **17**

3.1 Introduction 17

3.2 Recasting the problem 19

3.3 Aardvark: RBFT in action 21

3.4 Protocol description 26

3.4.1 Client request transmission 26

3.4.2 Replica agreement 31

3.4.3 Primary view changes . . . 36
3.4.4 Implementation . . . 39
3.5 Analysis . . . 39
3.6 Experimental evaluation . . . 42
3.6.1 Common case performance . . . 43
3.6.2 Evaluating faulty systems . . . 46
3.7 Conclusion . . . 52

Chapter 4 UpRight RSM Architecture **54**
4.1 UpRight architecture . . . 56
4.2 Division of responsibilities . . . 58
4.2.1 Library properties . . . 58
4.2.2 Application requirements . . . 60
4.3 Looking forward . . . 62

Chapter 5 UpRight Stages **63**
5.1 Basic stage interactions . . . 66
5.1.1 Client properties . . . 67
5.1.2 Authentication properties . . . 67
5.1.3 Order properties . . . 68
5.1.4 Execution properties . . . 70
5.1.5 Putting the stages together . . . 71
5.2 Network efficiency . . . 74
5.3 Garbage collection and transient crashes . . . 76
5.3.1 Order stage . . . 77
5.3.2 Execution stage. . . . 82
5.3.3 Authentication stage . . . 85
5.3.4 Client . . . 89
5.4 Full property list . . . 89
5.4.1 Client Properties . . . 89
5.4.2 Authentication stage properties . . . 90
5.4.3 Order stage properties . . . 91
5.4.4 Execution stage properties . . . 92
5.5 Supported optimizations . . . 93

5.6 Messages and notation 94
5.7 Stage level pseudo-code 96
5.7.1 Client operation 97
5.7.2 Authentication operation 97
5.7.3 Order operation 101
5.7.4 Execution operation 104
5.8 Conclusion 107

Chapter 6 UpRight Replication **108**
6.1 Consensus background 110
6.2 Replicated order stage 112
6.2.1 Normal-operation—Zyzzyvark 113
6.2.2 Checkpoint-operation 120
6.2.3 Interactions with other stages 122
6.2.4 Order stage properties 123
6.3 Replicated execution stage 126
6.3.1 Execution consensus 126
6.3.2 Execution-stage checkpoints 128
6.3.3 Interactions with other stages 133
6.3.4 Execution stage properties 135
6.4 Replicating authentication stage 136
6.4.1 Authentication consensus 137
6.4.2 Interactions with other stages 139
6.4.3 Authentication stage properties 140
6.5 Implementation and performance 141
6.6 Discussion 148
6.7 Conclusion 149

Chapter 7 UpRight Applications **150**
7.1 Request Processing 154
7.2 Checkpoint Generation 155
7.3 HDFS case study 160
7.3.1 Baseline system 160
7.3.2 UpRight-HDFS 162

7.3.3 Evaluation . . . 164
7.3.4 MapReduce . . . 167
7.4 ZooKeeper case study . . . 167
7.4.1 Baseline system . . . 168
7.4.2 UpRight-ZooKeeper . . . 169
7.4.3 Evaluation . . . 169
7.5 Conclusion and Discussion . . . 173

Chapter 8 Background and state machine replication **175**
8.1 RSM approach . . . 175
8.2 Consensus . . . 176
8.3 Recent RSM history . . . 176
8.4 Performance with failures . . . 177
8.5 Application fault tolerance . . . 178

Chapter 9 Conclusion **180**

Appendix A UpRight Library Byte Specifications **182**
A.1 Basic Message Structure . . . 182
A.2 Inter-stage messages . . . 186
A.2.1 Message Tags . . . 186
A.2.2 Inter-stage messages . . . 186
A.2.3 Order stage checkpoint . . . 194
A.3 Execution node specifications . . . 194
A.3.1 Message Tags . . . 194
A.3.2 Execution checkpoints . . . 196
A.3.3 Execution Messages . . . 196

Appendix B UpRight Library API **200**
B.1 Client API . . . 200
B.2 Server API . . . 202

Bibliography **205**

Vita **215**

List of Tables

2.1 (a) Acceptors required to solve asynchronous consensus under various failure models. c is the maximum number of crash failures and b is the maximum number of Byzantine failures tolerated while ensuring the system is both safe and live. u is the maximum number of failures tolerated while ensuring the system is up. r is the maximum number of commission failures tolerated while ensuring the system is right. (b) Acceptors required to solve asynchronous consensus under the crash (Byzantine) failure model for various values of $f = b = c$. (c) Acceptors required to solve asynchronous consensus under a hybrid failure model with varying values of b and c. (d) Acceptors required to solve asynchronous consensus under the UpRight model with varying values of u and r. Values representing equivalent configurations across tables are marked with emphasis (*italicized* for BFT configurations, **bolded** for CFT configurations, or underlined for HFT configurations). 14

3.1 Observed peak throughput of BFT systems in a fault-free case and when a single faulty client submits a carefully crafted series of requests. We detail our measurements in Section 3.6.2. † The result reported for Q/U is for correct clients issuing conflicting requests. ‡ The HQ prototype demonstrates fault-free performance and does not implement many of the error-handling steps required to resolve inconsistent MACs. 18

3.2 Peak throughput of Aardvark and PBFT for different implementation choices. 45

3.3 Observed peak throughput of BFT systems in the fault free case and under heavy client retransmission load. UDP network flooding corresponds to a single faulty client sending 9KB messages. TCP network flooding corresponds to a single faulty client sending requests to open TCP connections and is shown for TCP based systems. . . . 50
3.4 Throughput during intervals in which the primary delays sending PRE-PREPARE message (or equivalent) by 1, 10, and 100 ms. 51
3.5 Average throughput for a starved client that is shunned by a faulty primary versus the average per-client throughput for any other client. 51
3.6 Observed peak throughput and observed throughput when one replica floods the network with messages. UDP flooding consists of a replica sending 9KB messages to other replicas rather than following the protocol. TCP flooding consists of a replica repeatedly attempting to open TCP connections on other replicas. 52

5.1 Message specification for messages exchanged between stages. The sender and recipients of the messages are indicated. 95
5.2 Summary of symbols used and their meanings. 95

6.1 Summary of stage-level replication requirements. 109
6.2 Consensus semantics for messages related to the order stage. Each proposal or learn message is part of a single consensus instance. The utility messages are used by both consensus protocols. 122
6.3 State management messages exchanged between execution replicas. . 129
6.4 Summary of replication requirements for different checkpoint storage strategies. 134
6.5 Inter stage messages and their role in the execution consensus protocol. 134
6.6 Inter stage messages related to stage management. 135
6.7 Messages sent to and from the authentication stage. 139

7.1 Informal statement of application requirements. 151

A.1 Message Tags for all intra-node messages. 186
A.2 Set of messages for intra-node communication 194

List of Figures

2.1 Different classifications of failure types. (a) represents crash failures. (b) represent omission failures, a superset of crash failures. (c) represents Byzantine, or arbitrary, failures which encompass all behaviors. (d) represents commission failures, the set of Byzantine behaviors that cannot be classified as omission failures. 7

3.1 Physical network in Aardvark. 23
3.2 Architecture of a single replica. The replica utilizes a separate NIC for communicating with each other replica and a final NIC to communicate with the collection of clients. Messages from each NIC are placed on separate worker queues. 25
3.3 Basic communication pattern in Aardvark. 26
3.4 Decision tree followed by replicas while verifying a client request. The narrowing width of the relative volume of client requests that survive each step of the verification process. 28
3.5 Decision tree followed by a replica when handling messages received from another replica. The width of the edges indicates the rate at which messages reach various stages in the processing. 33
3.6 Average per request latency vs. average throughput for Aardvark, HQ, PBFT, Q/U, and Zyzzyva. 44
3.7 The latency of an individual client's requests running Aardvark with 210 total clients. The sporadic jumps represent view changes in the protocol. 46
3.8 CDF of request latencies for 210 clients issuing 100,000 requests with Aardvark servers. 47

4.1 Basic flow of messages in the UpRight architecture. 57

5.1 Message flow between idealized stages in the UpRight architecture. . 66
5.2 Messages exchanged between stages. (1) Clients send requests to the authentication stage. (2) The authentication stage sends validated request hashes to the order stage. (3) The order stage sends ordered batches to the execution stage. (4a, 4b) The execution stage fetches request bodies from the authentication stage. (4c) The execution stage sends responses to the clients. Note that the messages travel through the system in a clockwise fashion. 75
5.3 Interactions between persistent state at each stage. The state maintained by the other stages depends on the state maintained at the order stage. The order stage maintains one or two checkpoints and between CP_{interval} and $2 \times CP_{\text{interval}} - 1$ ordered batches. The authentication stage maintains every request referenced by an ordered batch stored at the order stage and at most one pending request per client. The execution stage maintains two checkpoints that correspond to order stage checkpoints. Additional details on the contents of the order and execution checkpoints can be found in Figure 5.4 and Figure 5.5 respectively. 78
5.4 Order stage checkpoint. 80
5.5 Execution stage checkpoint. 83
5.6 Pseudo-Code for the client 97
5.7 Pseudo-Code for the authentication stage to follow. 99
5.8 Pseudo-Code for the order stage to follow. 102
5.9 Pseudo-Code for the execution node to follow. 105

6.1 Basic communication pattern for complete agreement. 116
6.2 Basic communication pattern for tentative agreement. 117
6.3 Basic communication pattern for speculative agreement. 117
6.4 Basic communication pattern for the order stage checkpoint consensus protocol. Note that while the execution stage acts as a single proposer, each individual replica is a distinct learner. In the context of the UpRight library, learning is done only when a network or node failure occurs. 121

6.5 Execution consensus. 127
6.6 Execution replica pseudo-code related to intra-stage checkpoint and state transfer. 129
6.7 Authentication consensus. 138
6.8 Latency v. throughput for J-Zyzzyvark and JSZyzzyvark. 142
6.9 Latency v. throughput for JSZyzzyvark configured for various values of r and u. 143
6.10 Latency v. throughput for JSZyzzyvark configured for various values of r and u with authentication, order, and execution replicas colocated. 144
6.11 Jiffies per request. RQ indicates the jiffies at the authentication stage; Order indicates the jiffies at the order stage; Execution indicates the jiffies at the execution stage. 144
6.12 JSZyzzyvark performance when using the authentication replica and matrix signatures, standard signatures, and MAC authenticators. (1B requests) 146
6.13 JSZyzzyvark performance for 1B, 1KB, and 10KB requests, and for 1KB and 10KB requests where full requests, rather than digests, are routed through order replicas. 147

7.1 UpRight application architecture from an application developer perspective. The UpRight library is a black box with a well defined interface. At both the client and the server, the developer implements application-specific glue that connects the library shim to the original application. 153
7.2 The checkpoint/delta approach for managing application checkpoints. Original application checkpoints are taken infrequently, but the library requests a checkpoint every 100 batches. (a) shows the original application checkpoint taken after executing batch n. (b) shows the checkpoint returned to the replication library after executing batch $n + 100$. This checkpoint consists of the application checkpoint at n and the log of the next 100 batches. (c) shows the checkpoint returned to the replication library after executing batch $n+200$. (d) shows the checkpoint returned to the replication library after executing batch $n + 400$. 158

7.3 Checkpoint-deltas returned to the application. Each returned checkpoint-delta consists of a coarse grained application checkpoint and sufficient deltas to produce the next coarse grained checkpoint. 159
7.4 Throughput for HDFS and UpRight-HDFS. 165
7.5 CPU consumption (jiffies per GB of data read or written) for HDFS and UpRight-HDFS. 165
7.6 Completion time for requests issued by a single client. In (a), the HDFS NameNode fails and is unable to recover. In (b), a single UpRight-HDFS NameNode fails, and the system continues correctly. 166
7.7 Execution time for TeraGen and TeraSort MapReduce workloads. . . 168
7.8 Throughput for UpRight-ZooKeeper and ZooKeeper for workloads comprising different mixes of 1KB reads and writes. 170
7.9 Per-request CPU consumption for UpRight-ZooKeeper and ZooKeeper for a write-only workload. The y axis is in jiffies. In our system, one jiffy is 4 ms of CPU consumption. 172
7.10 Performance v. time as machines crash and recover for ZooKeeper and UpRight-ZooKeeper. 173

A.1 Messages are built upon a verified message base. This basis byte structure contains 4 fields: tag, payload size, payload, authentication 183
A.2 Basic byte structure of a message with simple MAC authentication. 184
A.3 Byte definition for a message authenticated with a MAC array. The sender is the replica responsible for generating the MACs, the Digest field is a digest of the tag, payload size, and sender fields. The MACs are generated using the byte representation of the digest rather than the full message. 184
A.4 Message authenticated with a matrix signature. The authentiation block of these messages consists of a collection of MAC Arrays that each authenticate the tag, size and payload. 185
A.5 Byte Specification of the Entry at the core of every request. 187
A.6 Byte Specification of the payload of a $\langle$AUTH-REQ, $\langle$REQ-CORE, c, n_c, hash(OP)$\rangle_{\vec{\mu}_{f,\mathcal{O}}}$, $f\rangle_{\mu_{f,o}}$ message. 188
A.7 Byte Specification of the payload of a $\langle \text{COMMAND}, n_o, c, n_c, \text{OP}, f\rangle_{\mu_{f,e}}$ message. 188

A.8 Byte Specification of a $\langle \text{NEXT-BATCH}, v, n_o, \mathcal{H}, \mathcal{B}, t, \text{bool}, o \rangle_{\vec{\mu}_{o,\mathcal{E}}}$ message 189
A.9 Byte encoding of non-determinism. The two fields correspond to time and a seed for random number generation. 190
A.10 Byte Specification of the $\langle \text{REPLY}, n_c, \mathcal{R}, \mathcal{H}, e, \rangle_{\mu_{e,c}}$ message. 190
A.11 Byte Specification of the payload for a $\langle \text{REQUEST-CP}, n_o, o \rangle_{\vec{\mu}_{o,\mathcal{E}}}$ message. 190
A.12 Byte Specification of the payload for a $\langle \text{RELEASE-CP}, \mathcal{T}_{\text{cp}}, n_o, o \rangle_{\vec{\mu}_{o,\mathcal{E}}}$ message. 191
A.13 Byte Specification of the payload for a $\langle \text{RETRANSMIT}, c, o, \vec{\mu}_{o,\mathcal{E}} \rangle_m$essage. 191
A.14 Byte Specification of the payload for a $\langle \text{LOAD-CP}, \mathcal{T}_{\text{cp}}, n_o, o \rangle_{\mu_{o,e}}$ message. 191
A.15 Byte specification of a $\langle \text{BATCH-COMPLETE}, v, n_o, \mathcal{C}, e \rangle_{\vec{\mu}_{e,\mathcal{F}}}$ message. . 192
A.16 Byte specification of a $\langle \text{FETCH}, n_o, c, n_c, \text{hash}(\text{OP}), e \rangle_{\vec{\mu}_{e,\mathcal{F}}}$ message. . . 192
A.17 Byte specification of a $\langle \text{CP-UP}, n_o, \mathcal{C}, e \rangle_{\vec{\mu}_{e,\mathcal{F}}}$ message. 193
A.18 Byte Specification of $\langle \text{LAST-EXEC}, n_e, e \rangle_{\vec{\mu}_{e,\mathcal{O}}}$ and $\langle \text{CP-LOADED}, n_o, e \rangle_{\vec{\mu}_{e,\mathcal{O}}}$ messages. 193
A.19 Byte specification for the payload of a $\langle \text{CP-TOKEN}, n_o, \mathcal{T}_{\text{cp}}, e \rangle_{\vec{\mu}_{e,\mathcal{O}}}$ message. 194
A.20 Order node checkpoint. 195
A.21 Order node checkpoint byte specification. 195
A.22 Exec node checkpoint. 196
A.23 Order node checkpoint byte specification. 197
A.24 Byte Specification of the payload of a $\langle \text{FETCH-EXEC-CP}, n, e \rangle_{\vec{\mu}_{e,\mathcal{E}}}$ message. 197
A.25 Byte Specification of the payload of a $\langle \text{EXEC-CP-STATE}, n, \mathcal{S}, e \rangle_{\mu_{e,e'}}$ message. 198
A.26 Byte Specification of the payload of a $\langle \text{FETCH-STATE}, \mathcal{T}_{\text{state}}, e \rangle_{\vec{\mu}_{e,\mathcal{E}}}$ message. 198
A.27 Byte Specification of the payload of a $\langle \text{STATE}, \mathcal{T}_{\text{state}}, \mathcal{S}, e \rangle_{\mu_{e,e'}}$ message. 199

B.1 Interface exported by the UpRight library to the application client. . 201
B.2 Interface implemented by the application client. 201
B.3 Interface implemented by the application server and called by the UpRight library. The six functions can be considered as three pairs of common functionality: (a) request execution, (b) checkpoint management, and (c) state transfer. 203

B.4 Interface exported by the UpRight library to the application server as call-backs. The functions can be considered in groups based on common functionality: (a) response processing, (b) checkpoint management, (c) state transfer, and (d) generic management. 204

Chapter 1

Introduction

Experiences with computer systems indicate an inconvenient truth: computers fail and they fail in interesting ways. Although using redundancy to protect against fail-stop failures is common practice [12, 19, 39, 44, 89, 108], non-fail-stop computer and network failures occur for a variety of reasons including power outage [51], disk or memory corruption [8, 90, 91], NIC malfunction [2, 21, 96], user error [41, 78], operating system and application bugs [82, 105, 106] or misconfiguration [71, 102], and many others. The impact of these failures can be dramatic, ranging from service unavailability [97] to stranding airplane passengers on the runway [21] to companies closing [14].

While high-stakes embedded systems have adopted Byzantine fault tolerant techniques (e.g., avionics [10, 31, 46]), general purpose computing continues to rely on techniques that are fundamentally crash tolerant. In a general purpose environment, the current best practices response to non-fail-stop failures can charitably be described as pragmatic: identify a root cause and add check-sums to detect the error and prevent it from causing more problems in the future. Pragmatic responses have proven effective for patching holes and protecting against faults once they have occurred; unfortunately the initial damage has already been done, and it is difficult to say if the patches made to address previous faults will protect against future failures.

We posit that an end-to-end solution based on Byzantine fault tolerant (BFT) state machine replication is an efficient and deployable alternative to current *ad hoc* approaches favored in general purpose computing. The replicated state machine

approach ensures that multiple copies of the same deterministic application execute requests in the same order and provides end-to-end assurance that independent transient failures will not lead to unavailability or incorrect responses. An efficient and effective end-to-end solution covers faults that have already been observed as well as failures that have not yet occurred, and it provides structural confidence that developers won't have to track down yet another failure caused by some unpredicted memory, disk, network, or other behavior.

While the promise of end-to-end failure protection is intriguing, significant technical and practical challenges currently prevent adoption in general purpose computing environments. On the technical side, it is important that end-to-end solutions maintain the performance characteristics of deployed systems: if end-to-end solutions dramatically increase computing requirements, dramatically reduce throughput, or dramatically increase latency during normal operation, then end-to-end techniques are not appealing. On the practical side, it is important that end-to-end approaches be both comprehensible and easy to incorporate: if the cost of end-to-end solutions is rewriting an application or trusting intricate and arcane protocols, then end-to-end solutions will not be widely adopted.

The goal of this thesis is to make deploying Byzantine fault tolerant systems in a general purpose computing environment easier. To that end, the contributions of this thesis fall into three broad categories. First, we *re-define* what it means for a system to be fault tolerant. Second, we *re-architect* a (Byzantine) fault-tolerant library. Third, we *re-engineer* legacy applications to be Byzantine fault tolerant.

- **Re-defining the problem.** We restate what it means for systems to be fault tolerant in two fundamental ways. First, we embrace the UpRight model for counting failures and designing systems. Second, we advocate the design of fault tolerant systems that are *robust* to failures, i.e., systems that provide solid performance even when failures occur.

  Chapter 2 presents the UpRight failure model, an alternative formulation to the traditional Byzantine and crash failure models. The UpRight model has three distinct advantages over traditional failure models. First, it can express the traditional crash [11], Byzantine [79], and hybrid [98] fault models. Second, systems designed under the UpRight failure model provide the specified fault tolerance at the minimal replication cost. Third, designing systems under the

UpRight failure model makes the question of providing "Byzantine or crash fault tolerance" a deployment rather than an implementation question; the implementation question becomes whether it is appropriate to provide "fault tolerance or no fault tolerance?"

Chapter 3 presents the case for *robust fault tolerance* and demonstrates that robust (Byzantine) fault tolerant systems are feasible. Fault tolerant systems have traditionally been evaluated based on the throughput provided during failure-free executions and ignored the performance in the presence of failures. A side effect of this evaluation focus has been protocol designs and prototype implementations that can be rendered unusable by a single faulty client or server. We argue that fault tolerant systems should be expected to perform well during failure-ful executions and demonstrate that robust fault tolerant implementations are possible.

- **Re-architecting BFT.** We revisit the design of BFT systems in order to correct, combine, and refine a multitude of ideas that have been developed in the last decade. This portion of the thesis focuses on the design and implementation of the UpRight library. The contribution from this portion of the thesis rests with (a) the specification of responsibilities for the library and the application, (b) the stage-wise description of the steps required for state machine replication, and (c) the use of consensus to fully describe the interactions between nodes in the system.

  Chapter 4 lays the foundation for the subsequent chapters. Chapter 4 establishes (1) the basic interaction between the UpRight library and a replicated application and (2) a new architecture for state machine replication. The UpRight library delivers a linearized sequence of *batches* of requests—rather than individual *requests* delivered by previous systems—to the application for deterministic execution. This subtle shift in the objects delivered to the application provides the application with additional freedom with respect to processing requests. The UpRight architecture divides state machine replication around three core functions—request authentication, request ordering, and request execution—rather than the traditional two (request ordering and request execution) [88]. We present a replication architecture based on separating authentication, order, and execution into three distinct stages and an

abstract protocol for coordinating those stages.

Chapter 5 details the interactions *between* the stages identified in Chapter 4. Our work at the stage-level is geared towards providing an end-to-end protocol for correct stages to follow that fulfills the library requirements described in the previous chapter. We ensure that the stage-level protocol provides the appropriate end-to-end properties with correct stages despite faulty clients, an unreliable network with finite bandwidth, finite storage, and transient crashes (i.e. due to temporary power outages).

While Chapter 5 focuses on the interaction between stages, Chapter 6 discusses the replicated *implementation* of each each stage of the UpRight architecture. We base each stage on consensus. Even though our design for the authentication, order, and execution stages are each based on consensus, the protocols implementing each stage require different amounts of replication and different coordination between the replicas.

- **Re-engineering deployed applications.** We demonstrate that BFT replication techniques can be incorporated into existing applications with modest effort and without decimating performance. This work requires us to design the interface between replication libraries and applications to be minimally invasive to the application and also to test that design by integrating the library into deployed applications.

  Chapter 7 describes the interface between the UpRight library and applications and relates our experience incorporating the UpRight library into the Hadoop distributed file system (HDFS) and Zookeeper distributed coordination service. We take a pragmatic view of the interactions between the library and applications. For example, we prioritize using existing mechanisms, e.g., for checkpoint generation, over highly optimized and generic functionality in the library that may require extensive modification to the application to be useful. We find that we can provide UpRight versions of HDFS and ZooKeeper that offer competitive performance at only nominal development effort.

  Appendix A describes the byte specification for all messages exchanged and persistent state in our prototype of the UpRight library. Appendix B details the Java interfaces exported to the application client and server by the UpRight library.

Chapter 2

Failure models and fault tolerance

We want distributed systems to be *up* (live) and *right* (safe). Intuitively, a system is up if it processes every received request and right if processed requests are processed correctly. The variety of ways in which things can go wrong, however, makes building systems that are up and right challenging: networks can fail—by delaying, corrupting, or dropping messages—and nodes can misbehave—by failing to take a specified action or taking an arbitrary unspecified action. A *fault tolerant* system is designed to be up and right despite node and network failures.

In this thesis, we target distributed systems that are UpRight; an UpRight system is up (i.e., live) despite up to u Byzantine failures and right (i.e., safe) despite up to r commission failures. This definition of UpRight fault tolerance is layered with jargon and technical terms. To understand the practical implications of UpRight fault tolerance we must first understand the terminology and taxonomy of how computers and the network behave (Section 2.1) and the relationship between UpRight fault tolerance and traditional notions of crash and Byzantine fault tolerance (Section 2.2). We conclude this chapter with a brief discussion of the practical benefits of building systems to provide UpRight fault tolerance (Section 2.3) as opposed to traditional crash [87], Byzantine [61], or hybrid fault tolerance [98].

2.1 Classifying node and network behaviors

There is a clear dichotomy between nodes that are correct and nodes that are faulty. Correct nodes always follow a protocol specification faithfully while faulty nodes deviate from the specification in some way. Failures can take different forms, and fault tolerant protocols must be designed under some *failure model* that defines the failures the protocol is designed to tolerate. The rest of this section explores the definition of such failure models.

2.1.1 Faulty behaviors

The simplest type of failures is a *crash* failure. A replica exhibits a *crash* failure [87] if it permanently halts. Note that a node that "crashes" and is subsequently rebooted does not exhibit a crash failure because the "crash" is not permanent. A replica that fails to send or receive a subset of messages exhibits a *general omission* failure [80]. A replica that arbitrarily deviates from its specification exhibits a *Byzantine* failure [61]. These failure types form a simple hierarchy: every crash failure is an omission failure and every omission failure is a Byzantine failure.

The traditional failure hierarchy provides a well-defined classification for every type of failure, but does not provide a convenient label for an important and interesting subset of failures: Byzantine failures that are not omission failures. These failures are called *commission* failures [72]. Intuitively, a node exhibits a commission failure when it deviates from its specification by taking an unnecessary or incorrect action. This is in contrast with omission failures, which are marked by the failure to take an action. We present a graphical depiction of the relationship between crash, omission, Byzantine, and commission failures in Figures 2.1(a)-(d).

Differentiating between omission and commission failures allows us to identify precisely the behaviors that make tolerating Byzantine failures more expensive than tolerating omission failures.

2.1.2 Correct behaviors

Correct nodes follow their specification faithfully. Many fault tolerant systems rely on a threshold of correct nodes to ensure correct operation. Fulfilling this expectation can be difficult given the practical reality that a power outage can cause every

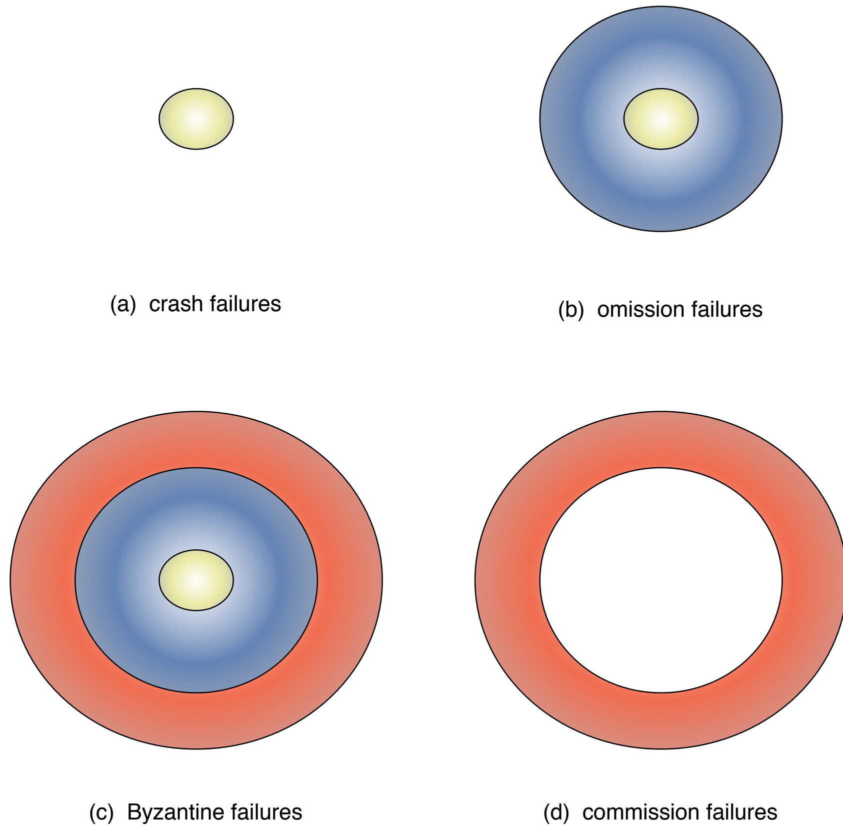

Figure 2.1: Different classifications of failure types. (a) represents crash failures. (b) represent omission failures, a superset of crash failures. (c) represents Byzantine, or arbitrary, failures which encompass all behaviors. (d) represents commission failures, the set of Byzantine behaviors that cannot be classified as omission failures.

machine in a data center to temporarily crash before power is restored. In theory, machines that exhibit *transient crash* behavior can be treated as "correct yet slow" and do not impact the safety guarantees provided by the system. In practice, ensuring that nodes remain "correct yet slow" despite transient crashes requires individual nodes to be engineered (a) to commit state to persistent memory before outputting messages onto the network and (b) to restore working state from persistent memory following a transient crash.

Note that a node that is not engineered to tolerate transient crashes may be technically guilty of a commission failure if it loses important state while recovering from a transient crash. Consider, for example, a banking service that loses all records of the last ten transactions, including a deposit of $\$10,000$ into a client's account, when it crashes due to a power outage. When power is restored and the service resumes operation it will have no record of the deposit and will incorrectly report the balance to be smaller than it should be. In this case the service is guilty of a commission failure—the client believes the transaction occurred but the service does not.

2.1.3 Cryptographic assumptions and notation

We assume that cryptographic techniques like collision-resistant hashing, message authentication codes (MACs), encryption, and signatures are secure. In particular, no node d can (a) create hash collisions or (b) forge the signature or MAC of correct node $c \neq d$. Note that any node can forge the signature or MAC of another node that has shared its authentication credentials; sharing authentication credentials constitutes a commission failure.

We denote a message X signed by principal p's public key as $\langle X \rangle_{\sigma_p}$. We denote a message X with a MAC appropriate for principals p and r as $\langle X \rangle_{\mu_{p,r}}$; by standard convention, the order of the nodes indicates that p is the sender and r is the recipient. We denote a message containing a *MAC authenticator*—an array of MACs appropriate for verification by multiple nodes—as $\langle X \rangle_{\vec{\mu}_p}$ or $\langle X \rangle_{\vec{\mu}_{p,\mathcal{R}}}$. The former notation denotes a message authenticated by principal p for verification by every node; the latter denotes a message authenticated by principal p for authentication by nodes in the set $\mathcal{R}$.

2.1.4 Network Behaviors

In this thesis, we focus on designing systems under the assumption that the network connecting nodes is *asynchronous* and *unreliable*. An asynchronous network provides no bound on how long after message is sent by a correct node it is received by the recipient. An unreliable network may arbitrarily reorder, lose, duplicate, or corrupt messages.

We define a *synchronous interval* [18, 33, 53], to be a period in which the network reliably delivers messages with a bounded delay.

Definition 1 (Synchronous interval)**.** *During a* synchronous interval *any message sent between correct nodes is delivered within a bounded delay T if the sender retransmits according to some schedule until the message is delivered.*

We assume that synchronous intervals of arbitrary length occur infinitely often. This assumption is known as *eventual synchrony* [33].

2.2 Fault tolerance

Fault tolerant systems are designed to be *safe* and *live* despite failures. Intuitively, a system is live (aka up) if it provides a response to client requests and is safe (aka right) if all provided responses are correct. The number of nodes required to implement a fault tolerant system depends on the number and types of failures to be tolerated in addition to the targeted safety and liveness properties. The primary focus of this section is exploring four different ways to formulate the number and type of failures that the system tolerates—crash fault tolerance, Byzantine fault tolerance, hybrid fault tolerance, and UpRight fault tolerance. To make the discussion more concrete, we describe the replication requirements for asynchronous *consensus* [79] protocols under each fault tolerance formulation. A consensus protocol is at the core of every replicated state machine.

Consensus. We focus our discussion on Lamport's formulation of Paxos-style consensus [53, 54, 56], which is based on the assignment of each node in the system to at least one of three roles: proposers, acceptors, and learners. Proposers propose values to the system, acceptors coordinate in some way to choose a single proposed

value, and learners learn values that have been chosen. In this context, consensus is defined by three safety properties [56]:

- Only a value proposed by a proposer can be chosen.
- Only a single value is chosen.
- Non-faulty learners only learn chosen values.

and a single liveness property:

- Given a sufficiently long synchronous interval, if a non-faulty proposer proposes a value, then non-faulty learners eventually learn a value.

A fault tolerant consensus protocols is safe and live for any number of faulty proposers and learners and a bounded number of faulty acceptors[1]

Crash fault tolerance. *Crash fault tolerant* (CFT) protocols are guaranteed to be safe and live despite up to c crash failures. In practice, omission failures and a lossy network are indistinguishable from any participant in the system, so asynchronous CFT systems are effectively safe and live despite up to c omission failures [28]. In general, a total of at least $2c + 1$ acceptors are required to implement a CFT consensus protocol that is safe and live despite c crash/omission failures[53, 56][2].

Byzantine fault tolerance. Researchers have developed a multitude of *Byzantine fault tolerant* (BFT) protocols designed to be safe and live despite up to b Byzantine failures [1, 18, 24, 26, 49, 50, 92, 100, 104, 107]. In general, a total of at least $3b + 1$ acceptors are required to implement a BFT consensus protocol that is safe and live despite b Byzantine failures[56, 79][3].

Because every omission failure is also a Byzantine failure, Byzantine fault tolerant systems provide protection against a wider variety of failures than crash fault tolerant systems. This makes BFT techniques very powerful and flexible,

[1]Note that this claim does not violate the FLP impossibility result [35] due to the inclusion of the "sufficiently long synchronous interval" condition in the statement of liveness.

[2]As Lamport notes, there are very specific configurations of proposers, acceptors, and learners that require fewer acceptors. A total of $2c + 1$ acceptors is *always* sufficient to implement CFT consensus.

[3]As Lamport notes, there are very specific configurations of proposers, acceptors, and learners that require fewer nodes. A total of $3b+1$ acceptors is *always* sufficient to implement BFT consensus.

benefits that can come at a significant cost. Consider, for example, a system running CFT consensus configured to tolerate up to $c = 4$ crash failures requires $9 = 2 \times 4 + 1$ acceptors. If it is subsequently discovered that it is important to tolerate one additional failure, a commission failure, the system has to be transitioned to use BFT techniques requiring $16 = 3 \times (4 + 1) + 1$ acceptors.

Hybrid fault tolerance. Hybrid fault tolerance (HFT) [98] is a response to the trepidation over the high cost of transitioning from CFT to BFT techniques and the observation that failures that require BFT are relatively rare. HFT protocols are designed to be safe and live despite up to b Byzantine and c crash failures. A total of at least $3b + 2c + 1$ acceptors are required to implement an HFT consensus protocol [98].

Returning to the example above where the system needs to be safe and live despite $c = 4$ crash failures and $b = 1$ Byzantine (commission) failure, HFT consensus can be implemented using $12 = 3 \times 1 + 2 \times 4 + 1$ acceptors.

UpRight fault tolerance. In this thesis, we advocate UpRight fault tolerance [23], initially described by Lamport [55] and subsequently employed by others [1, 32]. UpRight fault tolerance is motivated by the reality that systems should be up (i.e., live) and right (i.e., safe) despite failures and the recognition that the replication requirements for these two concerns are separate. Under UpRight fault tolerance, systems are designed to be live despite up to u failures of *any* type and safe despite up to r commission failures.

Intuitively, UpRight systems provide the following guarantees: (1) as long as there are at most u failures, the system is guaranteed to respond and (2) as long as there are at most r commission failures, any response is guaranteed to be correct. We note that when $u < r$, UpRight systems do not guarantee a response when there are between $u + 1$ and r commission failures, inclusive, but do guarantee that any received response will be correct.

The formulation of UpRight fault tolerance can be initially difficult to internalize. As a simple primer, consider a pair of hypothetical distributed systems, one where $u = 3$ and $r = 1$ and a second where $u = 1$ and $r = 3$.

The first system is appropriate for environments where (a) crashes are much more common than commission failures or (b) a higher premium is placed on liveness

than on safety. This configuration is guaranteed to provide a response to any request as long as at most three servers are faulty. Further, any response is guaranteed to be correct as long as at most one server is guilty of a commission failure. If two servers have been hacked (an extreme version of a commission failure) and all other servers are correct, then a user is guaranteed to receive a response ($u = 3 > 2$), but that response is not guaranteed to be correct ($r = 1 < 2$). If there are four failures then a user is not guaranteed to receive a response; further if at least two of the failures are commission failures then the user is not guaranteed that any received response can be trusted.

The second system is appropriate for environments where (a) commission failures are more common than omission failures and/or (b) a higher premium is placed on safety than on liveness. This configuration is guaranteed to provide a response as long as at most one server fails in any way. If multiple servers fail, then a user is not guaranteed a response. However, if at most three servers are guilty of commission failures any response is guaranteed to be correct.

In general, a total of $2u + r + 1$ acceptors are required to implement an UpRight consensus protocol that is live despite up to u Byzantine failures and safe despite up to r commission failures [32, 56][4].

We now revisit the previous example intended to tolerate four crash failures and one commission failure. In UpRight parlance, the system is expected to be up despite up to $u = 4$ omission failures and right despite up to $r = 1$ commission failures. An UpRight fault tolerant consensus protocol can be implemented using $10 = 2 \times 4 + 1 \times 1 + 1$ acceptors.

Comparing replication requirements. Table 2.1(a) summarizes the formulas for the minimum number of acceptors required to implement CFT, BFT, HFT, and UpRight consensus protocols.

Table 2.1(b) shows the minimum number of acceptors to implement crash (Byzantine) fault tolerant consensus for various values of c (b).

Table 2.1(c) shows the minimum number of acceptors required to implement HFT consensus for various values of b and c. Note that the row where $b = 0$

[4] As Lamport notes, there are very specific configurations of proposers, acceptors, and learners that require fewer acceptors. A total of $2u + r + 1$ acceptors is *always* sufficient to implement UpRight consensus.

corresponds to crash fault tolerance and the column where $c = 0$ corresponds to traditional Byzantine fault tolerance.

Table 2.1(c) shows the minimum number of acceptors required to implement UpRight consensus for various values of b and c. Note that crash, Byzantine, and hybrid fault tolerance can all be expressed under the UpRight framework. The row where $r = 0$ is equivalent to configurations that are safe and live despite up to $c = u$ crash failures (aka crash fault tolerance); the diagonal where $u = r$ is equivalent to configurations that are safe and live despite up to $b = u = r$ Byzantine failures (aka Byzantine fault tolerance); the upper right quadrant is equivalent to configurations that are safe and live despite up to $b = r$ Byzantine and $c = u + r$ crash failures (aka hybrid fault tolerance).

The end-to-end impact of adapting the UpRight language for fault tolerance is a reduction in the number of acceptors required when compared to BFT and HFT consensus. Intuitively, BFT solutions to consensus require more replicas ($3b+1$) than UpRight solutions ($2u+r+1$) because the BFT solutions count every failure against the budgets for both u and r, even if only one of the b total failures is expected to be a commission failure. Similarly, HFT solutions to consensus require more replicas ($2c+3b+1$) than UpRight solutions because the Byzantine portion of the equation ($3b$) increases both the u and the r portions of the of the replication requirements even though it is needed *only* because of the commission failures captured by r.

2.3 Why UpRight?

The previous sections describe terminology for classifying and counting failures in fault tolerant systems that departs from the customary notions of crash and Byzantine fault tolerance. Although UpRight fault tolerance generalizes crash Byzantine, and hybrid fault tolerance, it is tempting to view the discussion as a theoretical novelty. We believe that UpRight fault tolerance is more than a novelty and is in fact the right framework to use when designing fault tolerant systems.

UpRight fault tolerance provides several advantages when compared to other fault tolerance frameworks:

- UpRight fault tolerance is flexible and allows system designers and administrators to configure systems with the minimum number of servers. Traditional

CFT	$2c+1$
BFT	$3b+1$
Hybrid	$2c+3b+1$
UpRight	$2u+r+1$

(a) Replication requirements

f	CFT	BFT
0	**1**	*1*
1	**3**	*4*
2	**5**	*7*
3	**7**	*10*

(b) CFT and BFT replication

$b \backslash c$	0	1	2	3
0	***1***	**3**	**5**	**7**
1	*4*	6	8	10
2	*7*	9	11	13
3	*10*	12	14	16

(c) Hybrid replication

$r \backslash u$	0	1	2	3
0	***1***	**3**	**5**	**7**
1	2	*4*	6	8
2	3	5	*7*	9
3	4	6	8	*10*

(d) UpRight replication

Table 2.1: (a) Acceptors required to solve asynchronous consensus under various failure models. c is the maximum number of crash failures and b is the maximum number of Byzantine failures tolerated while ensuring the system is both safe and live. u is the maximum number of failures tolerated while ensuring the system is up. r is the maximum number of commission failures tolerated while ensuring the system is right. (b) Acceptors required to solve asynchronous consensus under the crash (Byzantine) failure model for various values of $f = b = c$. (c) Acceptors required to solve asynchronous consensus under a hybrid failure model with varying values of b and c. (d) Acceptors required to solve asynchronous consensus under the UpRight model with varying values of u and r. Values representing equivalent configurations across tables are marked with emphasis (*italicized* for BFT configurations, **bolded** for CFT configurations, or underlined for HFT configurations).

fault tolerance constructs are limited: crash fault tolerant systems cannot tolerate commission failures that result from server malfunction while Byzantine and hybrid fault tolerance can unnecessarily increase the replication requirements of the system.

- UpRight fault tolerance is a generalization of crash, Byzantine, and hybrid fault tolerance. If crash, Byzantine, or hybrid fault tolerance does accurately capture the design requirements of a specific environment then those requirements can be efficiently expressed using the UpRight framework. Further, if those requirements change then the changes can be accounted within the UpRight framework by adjusting the values of u and r—there is no need to shift from crash to Byzantine or hybrid fault tolerant protocol configuration as the design goals and deployment requirements change. UpRight is the single framework for all of your fault tolerant needs.

There are two non-questions that are frequently asked whenever Byzantine fault tolerant systems are discussed. While these questions do not apply to the technical discussion of UpRight fault tolerance, it is important to address them.

Do Byzantine failures actually happen? This question is outside the scope of this chapter and this thesis. We claim that fault tolerant systems should be designed to be UpRight and that it is the responsibility of the administrators deploying the system to choose values of u and r that are appropriate for their deployment. Put another way, rather than choosing between crash or Byzantine fault tolerance, system designers should choose UpRight fault tolerance and leave the decision of the type of failures to tolerate to the users of the system.

Stronger claims about the frequency and impact of commission failures require extensive deployment and classification and analysis of observed failures. We note that preventing transient crashes from becoming commission failures, as discussed in Section 2.1.2, is non-trivial since it can be difficult to determine when a disk write is really complete [75].

Who cares about fault tolerance, the failure models are wrong because correlated failures do happen? This question is misguided. The failure model describes and classifies the types of failures that can occur. Different fault tolerance

criteria express different failure scenarios under which safety and/or liveness are desired. The different failure models discussed in this chapter provide a framework for discussing and designing systems: the system should be live despite up to u failures and safe despite up to r commission failures. In principle, there is no reason not to attempt to build systems that are safe and live despite $u = r = n - 1$ failures (where n is the total number of servers). Specific problems, i.e. definitions of safety and liveness, may require u and r to be smaller and/or fractions of the total number of servers. Solutions to the consensus problem referred to in this chapter, for example, require $n \geq 2u + r + 1$ servers, introducing a failure threshold significantly smaller than the total number of servers in the system. The concern with correlated failures is not connected to the failure model, but rather to the specific techniques employed. The rest of this thesis presents a better way to reason about and design state machine replication; it does not demonstrate that state machine replication is the right approach for solving any specific deployment challenge.

Chapter 3

Robust Performance

Prelude

While the previous chapter focuses on the framework for discussing fault tolerant systems, i.e. how are failures classified and counted, this chapter focuses on what it means for a system to *be* fault tolerant. Although the discussion is presented in terms of asynchronous Byzantine fault tolerant state machine replication, the conclusion is generally applicable to any fault tolerant system.

3.1 Introduction

This chapter is motivated by a simple observation: although recently developed BFT state machine replication protocols have driven the costs of BFT replication to remarkably low levels [1, 18, 26, 49], the reality is that they don't tolerate Byzantine faults very well. In fact, a single faulty client or server can render these systems effectively unusable by inflicting multiple orders of magnitude reductions in throughput and even long periods of complete unavailability. Performance degradations of such degree are at odds with what one would expect from a system that calls itself Byzantine fault tolerant—after all, if a single fault can render a system unavailable, can that system truly be said to tolerate failures?

To illustrate the problem, Table 3.1 shows the measured performance of a variety of systems both in the absence of failures and when a single faulty client submits a carefully crafted series of requests. As we show later, a wide range of other

behaviors—faulty primaries, recovering replicas, etc.—can have a similar impact. We believe that these collapses are byproducts of a single-minded focus on designing BFT protocols with ever more impressive best-case performance. While this focus is understandable—after years in which BFT replication was dismissed as too expensive to be practical, it was important to demonstrate that high-performance BFT is not an oxymoron—it has led to protocols whose complexity undermines robustness in two ways: (1) the protocols' *design* includes *fragile optimizations* that allow a faulty client or server to knock the system off the optimized execution path to expensive alternative paths and (2) the protocol *implementations* often fail to handle properly all of the intricate corner cases, so that the implementations are even more vulnerable than the protocols appear on paper.

The primary contribution of this chapter is to advocate a new approach, *robust BFT* (RBFT), to building BFT systems. Our goal is to change the way BFT systems are designed and implemented by shifting the focus from constructing high-strung systems that maximize best-case performance to constructing systems that offer good and predictable performance under the broadest possible set of circumstances—including when faults occur.

In Section 3.2 we elaborate on the need to rethink Byzantine fault tolerance and identify a set of design principles for RBFT systems. In Section 3.3 we present a systematic methodology for designing RBFT systems and an overview of the Aardvark RBFT prototype. In Section 3.4 we describe in detail the important components of the Aardvark protocol. In Section 3.5 we present an analysis

System	Peak Throughput	Faulty Client
PBFT [18]	61.7k	0
Q/U [1]	23.8k	0†
HQ [26]	7.6k	N/A‡
Zyzzyva [49]	66k	0
Aardvark	38.7k	38.7k

Table 3.1: Observed peak throughput of BFT systems in a fault-free case and when a single faulty client submits a carefully crafted series of requests. We detail our measurements in Section 3.6.2. † The result reported for Q/U is for correct clients issuing conflicting requests. ‡ The HQ prototype demonstrates fault-free performance and does not implement many of the error-handling steps required to resolve inconsistent MACs.

of Aardvark's expected performance. In Section 3.6 we present our experimental evaluation.

3.2 Recasting the problem

The foundation of modern BFT state machine replication rests on an impossibility result and on two principles that assist us in dealing with it. The impossibility result, of course, is FLP [35], which states that no solution to consensus can be both safe and live in an asynchronous systems if nodes can fail. The two principles, first applied by Lamport to his Paxos protocol [53], are at the core of Castro and Liskov's seminal work on PBFT [17]. The first states that synchrony must not be needed for safety: as long as a threshold of faulty servers is not exceeded, the replicated service must always produce linearizable executions, independent of whether the network loses, reorders, or arbitrarily delays messages. The second recognizes, given FLP, that synchrony must play a role in liveness: clients are guaranteed to receive replies to their requests only during intervals in which messages sent to correct nodes are received within some fixed (but potentially unknown) time interval from when they are sent.

Within these boundaries, the engineering of BFT protocols has embraced Lampson's well-known recommendation: "Handle normal and worst-case separately as a rule because the requirements for the two are quite different. The normal case must be fast. The worst-case must make some progress" [62]. Ever since PBFT, the design of BFT systems has then followed a predictable pattern: first, characterize what defines the normal (common) case; then, pull out all the stops to make the system perform well for that case. While different systems don't completely agree on what defines the common-case [42], on one point they are unanimous: the common-case includes only *gracious executions*, defined as follows:

Definition 2 (Gracious execution)**.** *An execution is gracious iff (a) the execution is synchronous with some implementation-dependent short bound on message delay and (b) all clients and servers behave correctly.*

The results of this approach have been spectacular. In 2007, Zyzzyva reported throughput of over 85,000 null requests per second [49], and subsequent protocols have improved on that mark [42, 93].

Despite these impressive results, we argue that a single minded focus on aggressively tuning BFT systems for the best-case of gracious execution, a practice that we have engaged in with relish [49], is increasingly misguided, dangerous, and even futile.

It is misguided, because it encourages the design and implementation of systems that fail to deliver on their basic promise: to tolerate Byzantine faults. While providing impressive throughput during gracious executions, today's high-performance BFT systems are content to provide weak liveness guarantees (e.g. "eventual progress") in the presence of Byzantine failures. Unfortunately, as we previewed in Table 3.1 and show in detail in Section 3.6.2, these guarantees are weak indeed. Although current BFT systems can *survive* Byzantine faults without compromising safety, we contend that a system that can be made completely unavailable by a simple Byzantine failure can hardly be said to *tolerate* Byzantine faults.

It is dangerous, because it encourages *fragile optimizations*. Fragile optimizations are harmful in two ways. First, as we will see in Section 3.6.2, they make it easier for a faulty client or server to knock the system off its hard-won optimized execution path and enter an alternative, much more expensive one. Second, they weigh down the system with subtle corner-cases, increasing the likelihood of buggy or incomplete implementations.

It is (increasingly) futile, because the race to optimize common-case performance has reached a point of diminishing return where many services' peak demands are already far under the best-case throughput offered by existing BFT replication protocols. For such systems, *good enough is good enough*, and further improvements in best-case agreement throughput will have little effect on end-to-end system performance.

In our view, a BFT system will be most useful if it provides acceptable and dependable performance across the broadest possible set of executions, including executions with Byzantine clients and servers. In particular, the temptation of fragile optimizations should be resisted: a BFT system should be designed around an execution path that has three properties: (1) it provides acceptable performance, (2) it is easy to implement, and (3) it is robust against Byzantine attempts to push the system away from it. Optimizations for the common-case should be accepted only as long as they don't endanger these properties.

FLP tells us that we cannot guarantee liveness in an asynchronous environment. This is no excuse to focus only on performance during gracious executions. In particular, there is no theoretical reason why BFT systems should not be expected to perform well in what we call *uncivil executions*:

Definition 3 (Uncivil execution)**.** *An execution is* uncivil *iff (a) the execution is synchronous with some implementation-dependent bound on message delay, (b) up to f servers and any number of clients are Byzantine, and (c) all remaining clients and servers are correct.*

Hence, we propose to build RBFT systems that provide adequate performance during uncivil executions. Although we recognize that this approach is likely to reduce the best-case performance, we believe that for a BFT system a limited reduction in peak throughput is usually preferable to the devastating loss of availability that we report in Table 3.1 and Section 3.6.2.

Increased robustness may come at effectively no additional cost as long as a service's peak demand is below the throughput achievable through RBFT design: as a data point, our Aardvark prototype reaches a peak throughput of 38.7k req/s.

Similarly, when systems have other bottlenecks, Amdahl's law limits the impact of changing the performance of agreement. For example, we report in Section 3.6 that PBFT can execute almost 62,000 null requests per second, suggesting that agreement consumes $16.1\mu s$ per request. If, rather than a null service, we replicate a service for which executing an average request consumes $100\mu s$ of processing time, then peak throughput with PBFT settles to about 8613 requests per second. For the same service, a protocol with twice the agreement overhead of PBFT (i.e., $32.2\mu s$ per request), would still achieve peak throughput of about 7564 requests/second: in this hypothetical example, doubling agreement overhead would reduce peak end-to-end throughput by about 12%.

3.3 Aardvark: RBFT in action

Aardvark is a new BFT system designed and implemented to be robust to failures. The Aardvark protocol consists of three stages: client request transmission, replica agreement, and primary view change. This is the same basic structure of PBFT [18] and its direct descendants [7, 49, 50, 104, 107], but revisited with the goal of achiev-

ing an execution path that satisfies the properties outlined in the previous section: acceptable performance, ease of implementation, and robustness against Byzantine disruptions. To avoid the pitfalls of fragile optimizations, we focus at each stage of the protocol on how faulty nodes, by varying both the nature and the rate of their actions and omissions, can limit the ability of correct nodes to perform in a timely fashion what the protocol requires of them. This systematic methodology leads us to the three main design differences between Aardvark and previous BFT systems: (1) signed client requests, (2) resource isolation, and (3) regular view changes.

Signed client requests. Aardvark clients use digital signatures to authenticate their requests. Digital signatures provide *non-repudiation* and ensure that all correct replicas make identical decisions about the validity of each client request, eliminating a number of expensive and tricky corner cases found in existing protocols that make use of weaker (though faster) message authentication code (MAC) authenticators [17] to authenticate client requests. The difficulty with utilizing MAC authenticators is that they do not provide the non-repudiation property of digital signatures—one node validating a MAC authenticator does not guarantee that any other nodes will validate that same authenticator [3].

As we mentioned in the introduction to this chapter, digital signatures are generally seen as too expensive to use. Aardvark uses them only for client requests, where it is possible to push the expensive act of generating the signature onto the client while leaving the servers with the less expensive verification operation[1]. Server initiated communication—primary-to-replica, replica-to-replica, and replica-to-client communication—relies on MAC authenticators. The quorum-driven nature of server-initiated communication ensures that f or fewer[2] faulty replicas are unable to force the system into undesirable execution paths.

Because of the additional costs associated with verifying signatures in place of MACs, Aardvark must guard against new denial-of-service attacks where the system receives a large numbers of requests with signatures that need to be verified.

[1]In developing the Aardvark prototype we explicitly assumed that clients are external entitities that are not controlled by the service provider. In this context, the service provider is not responsible for costs incurred by the clients. In retrospect, this assumption is not appropriate for many deployments and impacts our design of the UpRight library in Chapters 4- 6.

[2]Note that we target systems that are safe and live despite up to $f = u = r$ faulty replicas in this chapter.

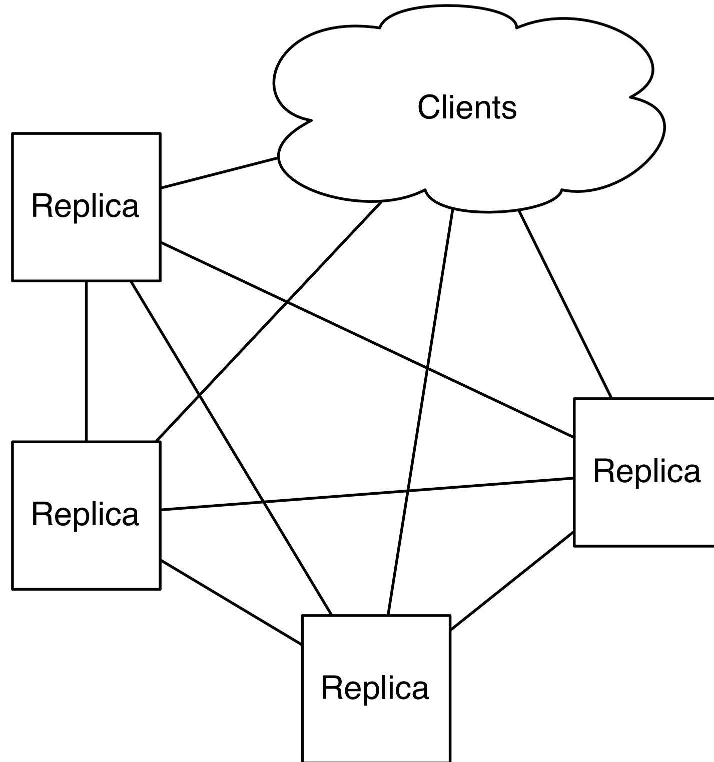

Figure 3.1: Physical network in Aardvark.

Our implementation limits the number of signature verifications a client can inflict on the system by (1) utilizing a hybrid MAC-signature construct to put a hard limit on the number of *faulty* signature verifications a client can inflict on the system and (2) forcing a client to complete one request before issuing the next.

Resource isolation. The Aardvark prototype implementation explicitly isolates network and computational resources.

As illustrated by Fig. 3.1, Aardvark uses separate network interface controllers (NICs) and wires to connect each pair of replicas. This step prevents a faulty server from interfering with the timely delivery of messages from good servers, as happened when a single broken NIC shut down the immigration system at the Los Angeles International Airport [21]. It also allows a node to defend itself against brute-force denial-of-service attacks by disabling the offending NIC. However, using

physically separate NICs for communication between each pair of servers incurs a performance cost, as Aardvark can no longer use ethernet multicast to optimize all-to-all communication, and limits the number of replicas in the system to the number of expansion slots on each machine.

As Figure 3.2 shows, Aardvark uses separate work queues for processing messages from clients and individual replicas. Employing a separate queue for client requests prevents client traffic from drowning out the replica-to-replica communications required for the system to make progress. Similarly, employing a separate queue for each replica allows Aardvark to schedule message processing fairly, ensuring that a replica is able to gather efficiently the quorums it needs to make progress. Aardvark can also easily leverage separate processors to process incoming client and replica requests. Taking advantage of hardware parallelism allows Aardvark to reclaim part of the costs paid to verify signatures on client requests.

We use simple brute-force techniques for resource scheduling. One could consider network-level scheduling techniques rather than distinct NICs in order to isolate network traffic and/or allow rate-limited multicast. Our goal is to make Aardvark as simple as possible, so we leave exploration of these techniques and optimizations for future work.

Regular view changes. To prevent a primary from achieving tenure and exerting absolute control on system throughput, Aardvark invokes the view change operation on a regular basis. Replicas monitor the performance of the current primary, slowly raising the required throughput level. If the current primary fails to provide the required throughput, replicas initiate a view change.

The key properties of this technique are:

1. During uncivil intervals, system throughput remains high even when replicas are faulty. Since a primary maintains its position only if it achieves some increasing level of throughput, Aardvark bounds throughput degradation caused by a faulty primary by either forcing the primary to be fast or selecting a new primary. When a new primary is selected, the required throughput is reset to an initial threshold, e.g. one half of the previous requirement.

2. As in prior systems, eventual progress is guaranteed when the system is eventually synchronous.

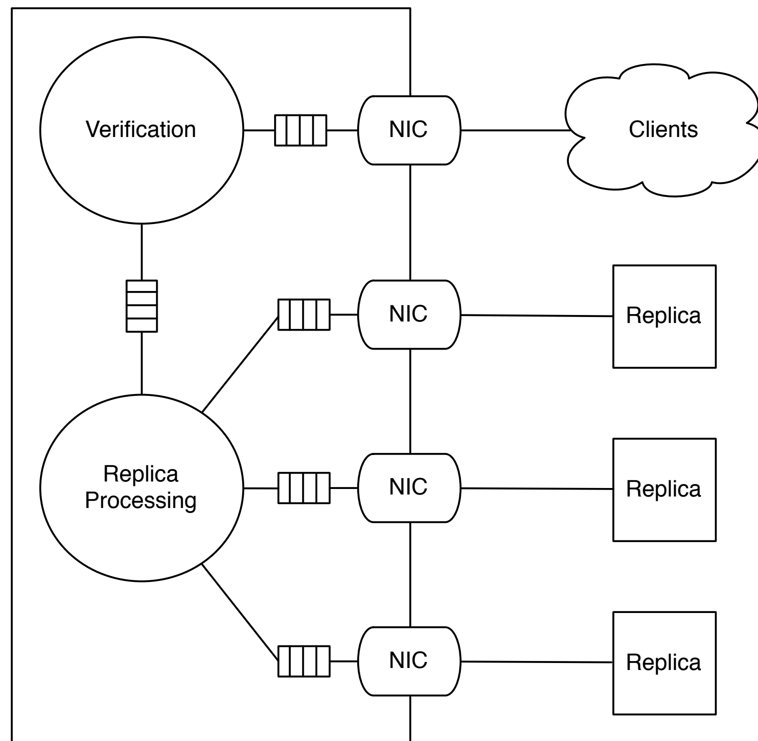

Figure 3.2: Architecture of a single replica. The replica utilizes a separate NIC for communicating with each other replica and a final NIC to communicate with the collection of clients. Messages from each NIC are placed on separate worker queues.

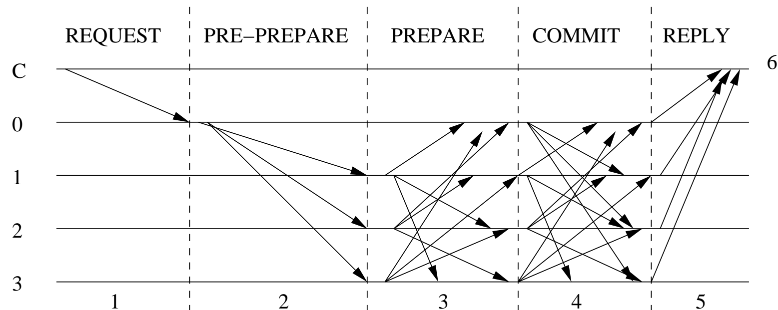

Figure 3.3: Basic communication pattern in Aardvark.

Previous systems have treated view change as an option of last resort that should only be used in desperate situations to avoid letting throughput drop to zero. However, although the phrase "view change" carries connotations of a complex and expensive protocol, in reality the cost of a view change is similar to the regular cost of agreement. Performing view changes regularly introduces short periods of time during which new requests are not being processed, but the benefits of rapidly evicting a misbehaving primary outweigh the periodic costs associated with performing view changes.

3.4 Protocol description

Figure 3.3 shows the agreement phase communication pattern that Aardvark shares with PBFT [18]. Variants of this pattern are employed in other recent BFT RSM protocols [1, 26, 42, 49, 93, 104, 107], and we believe that, just as Aardvark illustrates how the RBFT design approach can be applied to PBFT, new RBFT systems based on these other protocols can and should be constructed. We organize the following discussion around the numbered steps of the communication pattern of Figure 3.3.

3.4.1 Client request transmission

The fundamental challenge in transmitting client requests is ensuring that, upon receiving a client request, every replica comes to the same conclusion about the authenticity of the request. We ensure this property by having clients sign requests.

To guard against denial of service, we break the processing of a client request into a sequence of increasingly expensive steps. Each step serves as a filter, so that

more expensive steps are performed less often. For instance, we ask clients to include a MAC on their signed requests and have replicas verify only the signature of those requests whose MAC checks out. As mentioned in Section 3.3, Aardvark explicitly dedicates a single NIC to handling incoming client requests so that incoming client traffic does not interfere with replica-to-replica communication.

Protocol Description

The steps taken by an Aardvark replica to authenticate a client request follow.

1. Client sends a request to a replica.

A client c requests an operation o be performed by the replicated state machine by sending a request message $\langle\langle \text{REQUEST}, o, s, c\rangle_{\sigma_c},\ c\rangle_{\mu_{c,p}}$ to the replica p it believes to be the primary. If the client does not receive a timely response to that request, then the client retransmits the request $\langle\langle \text{REQUEST}, o, s, c\rangle_{\sigma_c},\ c\rangle_{\mu_{c,r}}$ to all replicas r. Note that the request contains the client sequence number s and is signed with signature σ_c. The signed message is then authenticated with a MAC $\mu_{c,r}$ for the intended recipient. The MAC ensures that the signature cannot be corrupted by an intermediary.

Upon receiving a client request, a replica proceeds to verify it by following a sequence of steps designed to limit the maximum load a client can place on a server, as illustrated by Figure 3.4:

(a) **Blacklist check.** If the sender c is not blacklisted, then proceed to step (b). Otherwise discard the message.

(b) **MAC check.** If $\mu_{c,p}$ is valid, then proceed to step (c). Otherwise discard the message.

(c) **Sequence check.** Compre the sequence number s_{req} of the most recently cached reply for client c to the sequence number s of the incoming request. If the request sequence number s is exactly $s_{cache} + 1$, then proceed to step (d). Otherwise

(c1) **Retransmission check.** Each replica uses an exponential back off to limit the rate of client reply retransmissions. If a reply has not been sent

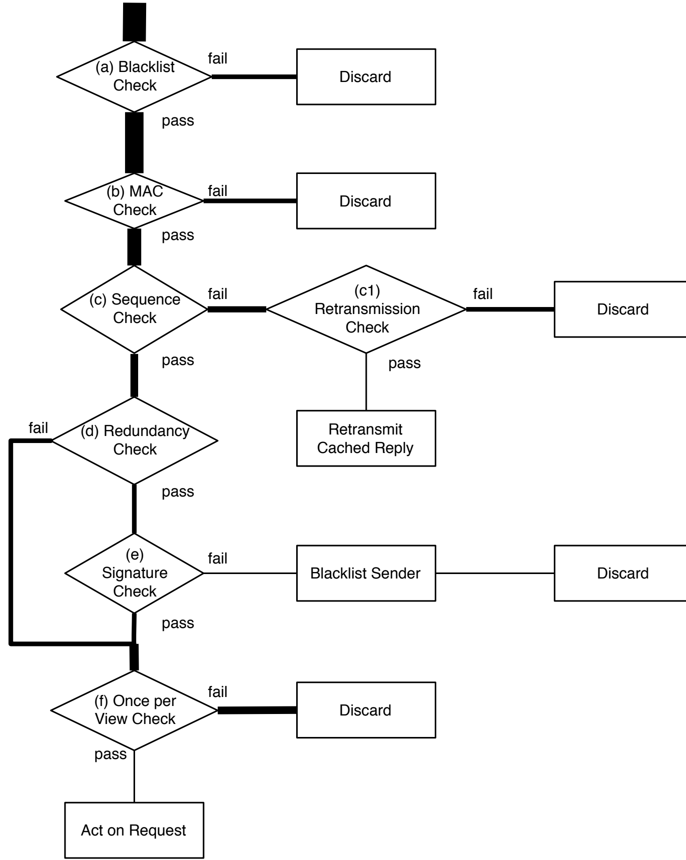

Figure 3.4: Decision tree followed by replicas while verifying a client request. The narrowing width of the relative volume of client requests that survive each step of the verification process.

to c recently, then retransmit the last reply sent to c. Otherwise discard the message.

(d) **Redundancy check.** Examine the most recent cached request from c. If no request from c with sequence number s_{req} has previously been verified or the request does not match the cached request, then proceed to step (e). Otherwise (the request matches the cached request from c) proceed to step (f).

(e) **Signature check.** If σ_c is valid, then proceed to step (f); additionally, if the request does not match the previously cached request for s_{req}, then blacklist c

and discard the message. Otherwise if σ_c is not valid, then blacklist the node x that authenticated $\mu_{x,p}$ and discard the message.

(f) **Once-per-view check.** If an identical request has been verified in a previous view, but not processed during the current view, then act on the request. Otherwise discard the message.

Primary and non-primary replicas act on requests in different ways. A primary adds requests to a PRE-PREPARE message that is part of the three-phase commit protocol described in Section 3.4.2. A non-primary replica r processes a request by authenticating the signed request with a MAC $\mu_{r,p}$ for the primary p and sending the message to the primary. Note that non-primary replicas will forward each request at most once per view, but they may forward a request multiple times provided that a view change occurs between each occurrence.

Note that a REQUEST message that is verified as authentic might contain an operation that the replicated service that runs above Aardvark rejects because of an access control list (ACL) or other service-specific security violation. From the point of view of Aardvark, such messages are valid and are delivered to all replicas in the same order. It is the responsibility of the replicated service to handle such messages and security violations, either by rejecting the operation at the service level or generating an application-level error code.

A node p only blacklists a sender c of a $\langle\langle \text{REQUEST}, o, s, c\rangle_{\sigma_c}, c\rangle_{\mu_{c,p}}$ message if (a) the MAC $\mu_{c,p}$ is valid but the signature σ_c is not or (b) the client applies the same sequence number to two distinct requests. A valid MAC is sufficient to ensure that routine message corruption is not the cause of the altered request or invalid signature sent by c, but rather that c has suffered a significant fault or is engaging in malicious behavior. A replica discards all messages it receives from a blacklisted sender and removes the sender from the blacklist after 10 minutes to allow reintegration of repaired machines.

Resource scheduling

Client requests are necessary to provide input to the RSM while replica-to-replica communication is necessary to process those requests. Aardvark implements separate work queues for receiving client requests and receiving replica-to-replica communication to limit the fraction of replica resources that clients are able to consume,

ensuring that a flood of client requests is unable to prevent replicas from making progress on requests already received. Of course, as in a non-BFT service, malicious clients can still deny service to other clients by flooding the network between clients and replicas. Defending against these attacks is an area of active independent research [63, 101].

We deploy our prototype implementation on dual-core machines. As Figure 3.2 shows, one core verifies client requests and the second runs the replica protocol. This explicit assignment allows us to isolate resources and take advantage of parallelism to partially mask the additional costs of signature verification.

Discussion

RBFT aims at minimizing the costs that faulty clients can impose on replicas. As Figure 3.4 shows, there are four actions triggered by the transmission of a client request that can consume significant replica resources: MAC verification (MAC check), retransmission of a cached reply, signature verification (signature check), and request processing (act on request). The cost a faulty client can cause increases as the request passes each successive check in the verification process, but the rate at which a faulty client can trigger this cost decreases at each step.

Starting from the final step of the decision tree, the design ensures that the most expensive message a client can send is a correct request as specified by the protocol, and it limits the rate at which a faulty client can trigger expensive signature checks and request processing to the maximum rate a correct client would. The sequence check step (c) ensures that a client can trigger signature verification or request processing for a new sequence number only after its previous request has been successfully executed. The redundancy check (d) prevents repeated signature verifications for the same sequence number by caching each client's most recent request. Finally, the once-per-view check (f) permits repeated processing of a request only across different views to ensure progress. The signature check (e) ensures that only requests that will be accepted by all correct replicas are processed. The net result of this filtering is that, for every k correct requests submitted by a client, each replica performs at most $k + 1$ signature verifications, and any client that imposes a $k + 1^{st}$ signature verification is blacklisted and unable to instigate additional signature verifications until it is removed from the blacklist.

Moving up the diagram, a replica responds to retransmission of completed requests paired with valid MACs by retransmitting the most recent reply sent to that client. The retransmission check (c1) imposes an exponential back-off on retransmissions, limiting the rate at which clients can force the replica to retransmit a response. To help a client learn the sequence number it should use, a replica resends the cached reply at this limited rate upon receipt of requests that are from the past and very far in the future.

Any request that fails the MAC check (b) is immediately discarded. MAC verifications occur on every incoming message that claims to have the right format unless the sender is blacklisted, in which case the blacklist check (a) results in the message being discarded. The rate of MAC verification operations is thus limited by the rate at which messages purportedly from non-blacklisted clients are pulled off the network, and the fraction of processing wasted is at most the fraction of incoming requests from faulty clients.

3.4.2 Replica agreement

Once a request has been transmitted from the client to the current primary, the replicas must agree on the request's position in the global order of operations. Aardvark replicas coordinate with each other using a standard three-phase-commit protocol [18].

The fundamental challenge in the agreement phase is ensuring that each replica can quickly collect the quorums of PREPARE and COMMIT messages necessary to make progress. Conditioning expensive operations on the gathering of a quorum of messages makes it easier to ensure robustness in two ways. First, it is possible to design the protocol so that incorrect messages sent by a faulty replica will never gain the support of a quorum of replicas. Second, as long as there exists a quorum of timely correct replicas, a faulty replica that sends correct messages too slowly, or not at all, cannot impede progress. Faulty replicas can introduce overhead also by sending useless message or by sending messages too quickly: to protect themselves, correct replicas in Aardvark process messages from other replicas in a round-robin fashion whenever messages from multiple replicas are available.

Not all expensive operations in Aardvark are triggered by a quorum. In particular, a correct replica that has fallen behind its peers may ask them for the

state it is missing by sending them a *catchup message* (see Section 3.4.2). Aardvark replicas defer processing such messages to idle periods. Note that this state-transfer procedure is self-tuning: if the system is unable to make progress because it cannot assemble quorums of PREPARE and COMMIT messages, then it will become idle and devote more time to processing catchup messages.

Agreement protocol

The agreement protocol requires replica-to-replica communication. A replica r filters, classifies, and finally acts on the messages it receives from another replica according to the decision tree shown in Figure 3.5:

(a) **Volume Check.** If replica q is sending too many messages, blacklist q and discard the message. Otherwise continue to step (b). Aardvark replicas use a distinct NIC for communicating with each replica. Using per-replica NICs allows an Aardvark replica to silence replicas that flood the network and impose excessive interrupt processing load. In our prototype, we disable a network connection when q's rate of message transmission in the current view is a factor of 20 higher than for any other replica. After disconnecting q for flooding, r reconnects q after 10 minutes, or when f other replicas are disconnected for flooding.

(b) **Round-Robin Scheduler.** Among the pending messages, select the next message to process from the available messages in round-robin order based on the sending replica ID. Discard received messages when the buffers are full.

(c) **MAC Check.** If the selected message has a valid MAC, then proceed to step (d) otherwise, discard the message.

(d) **Classify Message.** Classify the authenticated message according to its type:

- If the message is PRE-PREPARE, then process it immediately in protocol step 3 below.
- If the message is PREPARE or COMMIT, then add it to the appropriate quorum and proceed to step (e).
- If the message is a catchup message, then proceed to step (f).

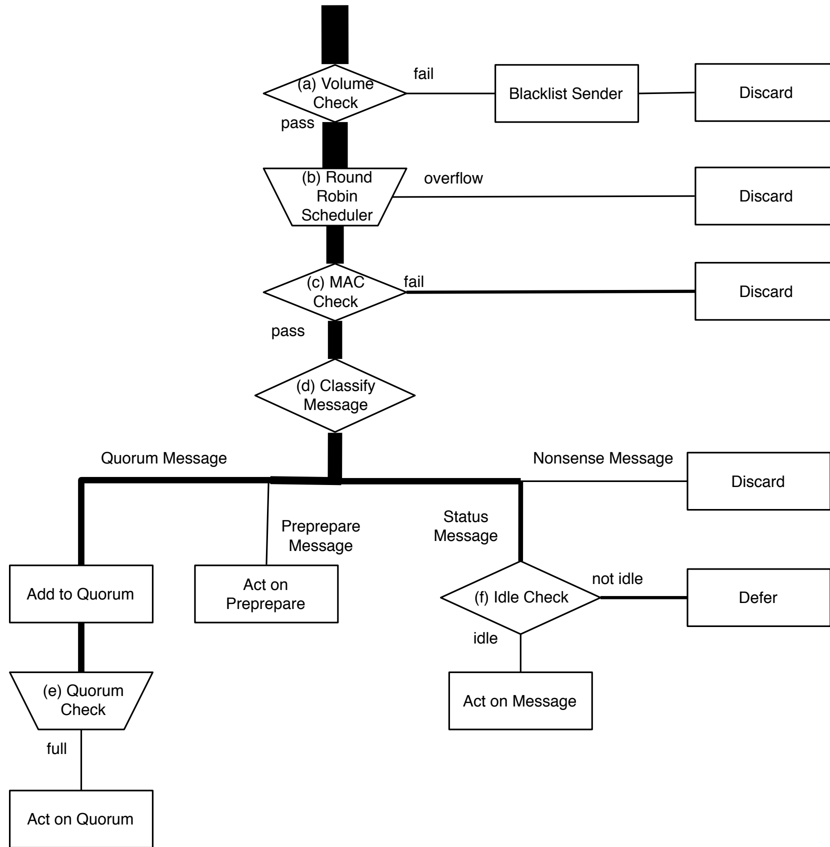

Figure 3.5: Decision tree followed by a replica when handling messages received from another replica. The width of the edges indicates the rate at which messages reach various stages in the processing.

- If the message is anything else, then discard the message.

(e) **Quorum Check.** If the quorum to which the message was added is complete, then act as appropriate in protocol steps 4-6 below.

(f) **Idle Check.** If the system has free cycles, then process the catchup message. Otherwise, defer processing until the system is idle.

Replica r applies the above steps to each message it receives from the network. Once messages are appropriately filtered and classified, the agreement protocol continues from step 2 of the communication pattern in Figure 3.3.

2. Primary forms a PRE-PREPARE message containing a set of valid requests and sends the PRE-PREPARE to all replicas.

The primary creates and transmits a $\langle$PRE-PREPARE, $v, n,$ $\langle$REQUEST, $o, s, c\rangle_{\sigma_c}\rangle_{\vec{\mu}_p}$ message where v is the current view number, n is the sequence number for the PRE-PREPARE, and the authenticator is valid for all replicas. Although we show a single request as part of the PRE-PREPARE message, multiple requests can be batched in a single PRE-PREPARE [18, 37, 49, 50].

3. Replica receives PRE-PREPARE from the primary, authenticates the PRE-PREPARE, and sends a PREPARE to all other replicas.

Upon receipt of $\langle$PRE-PREPARE, $v, n,$ $\langle$REQUEST, $o, s, c\rangle_{\sigma_c}\rangle_{\vec{\mu}_p}$ from primary p, replica r verifies the message's authenticity following a process similar to the one described in Section 3.4.1 for verifying requests. If r has already accepted the PRE-PREPARE message, r discards the message preemptively. If r has already processed a different PRE-PREPARE message with $n' = n$ during view v, then r discards the message. If r has not yet processed a PRE-PREPARE message for n during view v, r first checks that the appropriate portion of the MAC authenticator $\vec{\mu}_p$ is valid. If the replica has not already done so, it then checks the validity of σ_c. If the authenticator is not valid r discards the message. If the authenticator is valid and the client signature is invalid, then the replica blacklists the primary and requests a view change. If, on the other hand, the authenticator and signature are both valid, then the replica logs the PRE-PREPARE message and forms a $\langle$PREPARE, $v, n,$ h,$r\rangle_{\vec{\mu}_r}$ to be sent to all other replicas where h is the digest of the set of requests contained in the PRE-PREPARE message.

4. Replica receives $2f$ PREPARE messages that are consistent with the PRE-PREPARE message for sequence number n and sends a COMMIT message to all other replicas.

Following receipt of $2f$ matching PREPARE messages from non-primary replicas r' that are consistent with a PRE-PREPARE from primary p, replica r sends a $\langle$COMMIT,$v, n, r\rangle_{\vec{\mu}_r}$ message to all replicas. Note that the PRE-PREPARE message from the primary is the $2f + 1^{\text{st}}$ message in the PREPARE quorum.

5. Replica receives $2f + 1$ COMMIT messages, commits and executes the request, and sends a REPLY message to the client.

After receipt of $2f+1$ matching $\langle\text{COMMIT}, v, n, r'\rangle_{\vec{\mu}_{r'}}$ from distinct replicas r', replica r commits and executes the request before sending $\langle\text{REPLY}, v, u, r\rangle_{\mu_{r,c}}$ to client c where u is the result of executing the request and v is the current view.

6. The client receives $f+1$ matching REPLY messages and accepts the request as complete.

We also support Castro's tentative execution optimization [18]. The details of tentative execution do not impact the RBFT design and analysis.

Catchup messages. State catchup messages are not an intrinsic part of the agreement protocol, but they fulfill the important logistical priority of bringing replicas that have fallen behind back up to speed. If replica r receives a catchup message from a replica q that has fallen behind, then r sends q the state that q requires to catch up and resume normal operations. Sending catchup messages is vital to allow temporarily slow replicas to avoid becoming permanently non-responsive, but it also offers faulty replicas the chance to impose significant load on their non-faulty counterparts. Aardvark explicitly delays the processing of catchup messages until there are idle cycles available at a replica—as long as the system is making progress, processing a high volume of requests, there is no need to spend time bringing a slow replica up to speed!

Discussion

We now discuss the Aardvark agreement protocol through the lens of RBFT, starting from the bottom of Figure 3.5. Because every quorum contains at least a majority of correct replicas, faulty replicas can only marginally alter the rate at which correct replicas take actions (e) that require a quorum of messages. Further, because a correct replica processes catchup messages (f) only when otherwise idle, faulty replicas cannot use catchup messages to interfere with the processing of other messages. When client requests are pending, catchup messages are processed only if too many correct replicas have fallen behind and the processing of quorum messages needed for agreement has stalled—and only until enough correct replicas to enable progress have caught up. Also note that the queue of pending catchup messages is finite, and a replica discards excess catchup messages. If the number of discarded messages exceeds a fixed maximum, then clear the queue of pending catchup messages and

reset the discarded message count.

A replica processes PRE-PREPARE messages at the rate they are sent by the primary. If a faulty primary sends them too slowly or too quickly, throughput may be reduced, hastening the transition to a new primary as described in Section 3.4.3.

Finally, a faulty replica could simply bombard its correct peers with a high volume of messages that are eventually discarded. The round-robin scheduler (b) limits the damage that can result from this attack: if c of its peers have pending messages, then a correct replica wastes at most $\frac{1}{c}$ of the cycles spent checking MACs and classifying messages on what it receives from any faulty replica. The round-robin scheduler also discards messages that overflow a bounded buffer, and the volume check (a) similarly limits the rate at which a faulty replica can inject messages that the round-robin scheduler will eventually discard.

3.4.3 Primary view changes

Employing a primary to order requests enables batching [18, 37] and avoids the need to trust clients to obey a back-off protocol [1, 22]. However, because the primary is responsible for selecting which requests to execute, the system throughput is at most the throughput of the primary. The primary is thus in a unique position to control both overall system progress [4, 7] and fairness to individual clients.

The fundamental challenge to safeguarding performance against a faulty primary is that a wide range of primary behaviors can hurt performance. For example, the primary can delay processing requests, discard requests, corrupt clients' MAC authenticators, introduce gaps in the sequence-number space, unfairly delay or drop some clients' requests but not others, etc.

Hence, rather than designing specific mechanisms to defend against each of these threats, past BFT systems [18, 49] have relied on view changes to replace an unsatisfactory primary with a new, hopefully better, one. Past systems trigger view changes conservatively, only changing views when it becomes apparent that the current primary is unlikely to allow the system to make even minimal progress.

Aardvark includes the same view change mechanism and triggers described for PBFT [18]; in conjunction with the agreement protocol, view changes in PBFT are sufficient to ensure eventual progress. They are not, however, sufficient to ensure acceptable progress, so Aardvark adds additional *adaptive throughput* triggers

that can cause a view change when the current throughput is determined to be insufficient.

Adaptive throughput

Replicas monitor the throughput of the current primary. If a replica judges the primary's performance to be insufficient, then the replica initiates a view change. More specifically, replicas in Aardvark expect two things from the primary: a regular supply of PRE-PREPARE messages and high sustained throughput. Following the completion of a view change, each replica starts a heartbeat timer that is reset whenever the next valid PRE-PREPARE message is received. If a replica does not receive the next valid PRE-PREPARE message before the heartbeat timer expires, the replica initiates a view change. To ensure eventual progress, a correct replica doubles the heartbeat interval each time the timer expires. Once the timer is reset because a PRE-PREPARE message is received, the replica resets the heartbeat timer back to its initial value. The value of the heartbeat timer is application and environment specific: our implementation uses a heartbeat of 40ms, so that a system that tolerates f failures demands a minimum of 1 PRE-PREPARE every $2^f \times 40$ms during uncivil intervals.

The periodic checkpoints that, at pre-determined intervals, correct replicas must take to bound their state offer convenient synchronization points to assess the throughput that the primary is able to deliver. If the observed throughput in the interval between two successive checkpoints falls below a specified threshold, initially 90% of the maximum throughput observed during the previous n views, the replica initiates a view change to replace the current primary. At each checkpoint interval following an initial grace period at the beginning of each view, 5s in our prototype, the required throughput is increased by a factor of 0.01. Continually raising the bar that the current primary must reach in order to stay in power guarantees that a view change will eventually occur and replace the primary, restarting the process with the next primary. Conversely, if the system workload changes, the required throughput adjusts over n views to reflect the performance that a correct primary can provide. Note that every replica decides to initiate a view change independently, so some correct replicas may initiate a view change while others do not. As long as the remaining replicas are satisfied with the current throughput, they can continue

processing messages in the current view even though some replicas have stopped processing requests in their desire to join the next view.

The combined effect of Aardvark's new expectations on the primary is that during the first 5s of a view the primary is required to provide throughput of at least 1 request per 40ms or face eviction. The throughput of any view that lasts longer than 5s is at least 90% of the maximum throughput observed during the previous n views.

Fairness

In addition to hurting overall system throughput, primary replicas can influence which requests are processed. A faulty primary could be unfair to a specific client (or set of clients) by neglecting to order requests from that client. To limit the magnitude of this threat, replicas track fairness of request ordering. When a replica receives from a client a request that it has not seen in a PRE-PREPARE message, it adds the message to its request queue and, before forwarding the request to the primary, it records the sequence number k of the most recent PRE-PREPARE received during the current view. The replica monitors future PRE-PREPARE messages for that request, and if it receives two PRE-PREPAREs for another client before receiving a PREPARE for client c, then it declares the current primary to be unfair and initiates a view change. This ensures that two clients issuing comparable workloads observe throughput values within a constant factor of each other.

Discussion

The adaptive view change and PRE-PREPARE heartbeats leave a faulty primary with two options: it can provide substandard service and be replaced promptly, or it can remain the primary for an extended period of time and provide service comparable to what a non-faulty primary would provide. A faulty primary that does not make any progress will be caught very quickly by the heartbeat timer and summarily replaced. To avoid being replaced, a faulty primary must issue a steady stream of PRE-PREPARE messages until it reaches a checkpoint interval, when it is going to be replaced until it has provided the required throughput. To do *just* what is needed to keep ahead of its reckoning for as long as possible, a faulty primary will be forced to deliver 95% of the throughput expected from a correct primary.

Periodic view changes may appear to institutionalize overhead, but their cost is actually relatively small. Although the term *view change* evokes images of substantial restructuring, in reality a view change costs roughly as much as a single instance of agreement with respect to message/protocol complexity: when performed every 100+ requests, periodic view changes have marginal performance impact during gracious or uncivil intervals. We quantify these overheads experimentally in section 3.6.

3.4.4 Implementation

new section The Aardvark prototype is based on the December 2007 release of the PBFT code base [9]. Our implementation of Aardvark consists of selected modifications to implement the signed client requests, periodic view changes, and resource scheduling discussed above. We rely on the same basic data structures as PBFT and the UDP-based network communication provided by PBFT [9].

3.5 Analysis

In this section, we analyze the throughput characteristics of Aardvark when the number of client requests is large enough to saturate the system and a fraction g of those requests is correct. We show that Aardvark's throughput during long enough uncivil executions is within a constant factor of its throughput during gracious executions of the same length provided there are sufficient correct clients to saturate the servers.

For simplicity, we restrict our attention to a *simplified Aardvark* implementation on a single-core machine with a processor speed of κ GHz. We consider only the computational costs of the cryptographic operations—verifying signatures, generating MACs, and verifying MACs, requiring θ, α, and α cycles, respectively[3]. Since these operations occur only when a message is sent or received, and the cost of sending or receiving messages is small, we expect similar results when modeling network costs explicitly.

We begin by calculating simplified Aardvark's peak throughput during a gracious view, i.e. a view that occurs during a gracious execution, in Theorem 1. We

[3]Note that generating and verifying MACs are symmetric operations and have identical cost α.

then show in Theorem 2 that during uncivil views, i.e. views that occur during uncivil executions, with a correct primary simplified Aardvark's throughput is at least g times the throughput achieved during a gracious view; as long as the primary is correct, faulty replicas are unable to adversely impact simplified Aardvark's throughput. Finally, we show that the throughput of an uncivil execution is at least the fraction of correct replicas times g times the throughput achieved during a gracious view, in Theorem 3. Note that the latter two theorems are applicable only when the workload remains constant across multiple gracious views. The Aardvark adaptive view change mechanism can require up to n views to converge on the appropriate required throughput for a given workload.

We begin in Theorem 1 by computing t_{peak}, simplified Aardvark's peak throughput during a gracious view, i.e. a view that occurs during a gracious execution. We then show in Theorem 2 that during uncivil views in which the primary replica is correct, simplified Aardvark's peak throughput is only reduced to $g \times t_{peak}$: in other words, ignoring low level network overheads, faulty replicas are unable to curtail simplified Aardvark's throughput when the primary is correct. Finally, we show in Theorem 3 that the throughput across all views of an uncivil execution is within a constant factor of $\frac{n-f}{n} \times g \times t_{peak}$.

Theorem 1. *Consider a gracious view during which the system is saturated, all requests come from correct clients, and the primary generates batches of requests of size b. Simplified Aardvark's throughput is then at least* $\frac{\kappa}{\theta+\frac{(4n-2b-4)}{b}\alpha}$ *operations per second.*

Proof. We examine the actions required by each server to process one batch of size b. For each request in the batch, every server verifies one signature. The primary also verifies one MAC per request. For each batch, the primary generates $n-1$ MACs to send the PRE-PREPARE and verifies $n-1$ MACs upon receipt of the PREPARE messages; replicas instead verify one MAC in the primary's PRE-PREPARE , generate $(n-1)$ MACs when they send the PREPARE messages, and verify $(n-2)$ MACs when they receive them. Finally, each server first sends and then receives $n-1$ COMMIT messages, for which it generates and verifies a total of $n-2$ MACs, and generates a final MAC for each request in the batch to authenticate the response to the client. The total computational load per request is thus $\theta + \frac{(4n+2b-4)}{b}\alpha$ at the primary, and $\theta + \frac{(4n+b-4)}{b}\alpha$ at a replica. The system's throughput at saturation

during a sufficiently long view in a gracious interval is thus at least $\frac{\kappa}{\theta+\frac{(4n+2b-4)}{b}\alpha}$ requests/sec. □

Theorem 2. *Consider an uncivil view in which the primary is correct and at most f replicas are Byzantine. Suppose the system is saturated, but only a fraction of the requests received by the primary are correct. The throughput of simplified Aardvark in this uncivil view is within a constant factor of its throughput in a gracious view in which the primary uses the same batch size.*

Proof. Let θ and α denote the cost of verifying, respectively, a signature and a MAC. We show that if g is the fraction of correct requests then the throughput during uncivil views with a correct primary approaches g of the gracious view's throughput as the ratio $\frac{\alpha}{\theta}$ tends to 0.

In an uncivil view, faulty clients may send unfaithful requests to every server. Before being able to form a batch of b correct requests, the primary may have to verify $\frac{b}{g}$ signatures and MACs, and correct replicas may verify $\frac{b}{g}$ signatures and an additional $(\frac{b}{g})(1-g)$ MACs. Because a correct server processes messages from other servers in round-robin order, it will process at most two messages from a faulty server per message that it would have processed had the server been correct. The total computational load per request is thus $\frac{1}{g}(\theta+\frac{b(1+g)+4g(n-1+f)}{b}\alpha)$ at the primary, and $\frac{1}{g}(\theta+\frac{b+4g(n-1+f)}{b}\alpha)$ at a replica. The system's throughput at saturation during a sufficiently long view in an uncivil interval with a correct primary thus is at least $\frac{g\kappa}{\theta+\frac{(b(1+g)+4g(n-1+f)}{b}\alpha}$ requests per second: as the ratio $\frac{\alpha}{\theta}$ tends to 0, the ratio between the uncivil and gracious throughput approaches g. □

Theorem 3. *For sufficiently long uncivil executions and for small f, the throughput of simplified Aardvark, when properly configured, is within a constant factor of its throughput in a gracious execution in which primary replicas use the same batch size and the system is saturated.*

Proof. First consider the case in which all the uncivil views have correct primary replicas. Assume that in a properly configured simplified Aardvark, $t_{baseViewTimeout}$ is set so that during an uncivil interval, a view change to a correct primary completes within $t_{baseViewTimeout}$. Since a primary's view lasts at least $t_{gracePeriod}$, as the ratio $\frac{\alpha}{\theta}$ tends to 0, the ratio between the throughput during a gracious view and an uncivil interval approaches $g\frac{t_{gracePeriod}}{t_{baseViewTimeout}+t_{gracePeriod}}$

Now consider the general case. If the uncivil interval is long enough, at most $\frac{f}{n}$ of its views will have a Byzantine primary. Simplified Aardvark's heartbeat timer provides two guarantees. First, a Byzantine server that does not produce the throughput that is expected of a correct server will not last as primary for longer than a grace period. Second, a correct server is always retained as a primary for at least the length of a grace period. Furthermore, since the throughput expected of a primary at the beginning of a view is a constant fraction of the maximum throughput achieved by the primary replicas of the last n views, faulty primary replicas cannot arbitrarily lower the throughput expected of a new primary. Finally, since the view change timeout is reset after a view change that results in at least one request being executed in the new view, no view change attempt takes longer then $t_{maxViewTimeout} = 2^f t_{baseViewTimeout}$. It follows that, during a sufficiently long uncivil interval, the throughput will be within a factor of $\frac{t_{gracePeriod}}{t_{maxViewTimeout}+t_{gracePeriod}} \frac{n-f}{n}$ of that of Theorem 2, and, as $\frac{\alpha}{\theta}$ tends to 0, the ratio between the throughput during uncivil and gracious intervals approaches $g\frac{t_{gracePeriod}}{t_{maxViewTimeout}+t_{gracePeriod}} \frac{(n-f)}{n}$. □

3.6 Experimental evaluation

In evaluating Aardvark we compare the throughput and latency provided by Aardvark and previous replication protocol prototypes (PBFT [18], HQ [26], Q/U [1], and Zyzzyva [49]) during failure-free and failure-ful executions. Our evaluation demonstrates three points: (a) despite our choice to utilize signatures, change views regularly, and forsake IP multicast, Aardvark's peak throughput is competitive with that of existing systems ; (b) existing systems are vulnerable to significant disruption as a result of a broad range of Byzantine behaviors; and (c) Aardvark is robust to a wide range of Byzantine behaviors. When evaluating existing systems, we attempt to identify places where the prototype *implementation* departs from the published *protocol*.

Environment We evaluate the performance of Aardvark, PBFT [18, 9], HQ [26], Q/U [1, 83], and Zyzzyva [49] on an Emulab cluster [103] deployed at the University of Texas at Austin. This cluster consists of machines with 3GHz Intel Pentium 4 Xeon processors with hyperthreading, 1GB of memory, and 1 Gb/s Ethernet connections.

The code bases used to report our results are provided by the respective systems' authors. James Cowling provided us the December 2007 public release of the PBFT code base [9] as well as a copy of the HQ codebase. We used version 1.3 of the Q/U codebase, provided to us by Michael Abd-El-Malek in October 2008 [83]. The Zyzzyva codebase is the version used in the SOSP 2007 paper [49]. The Aardvark code is the version used for the NSDI 2009 paper [24]. We rely on the existing configurations for each system to handle $f = 1$ Byzantine failures.

Method. Our basic experimental setup involves correct clients that operate in a closed loop—that is they issue requests one at a time and do not issue request i until they receive a response to request $i-1$. Unless otherwise noted, correct clients issue 100k 4KB requests. We increase system load by increasing the number of clients. Clients record the time at which each request is issued and the response received. We calculate the average latency of all requests issued by all clients. We calculate per second throughput by dividing the total duration of the experiment, in seconds, by the total number of requests issued by all clients.

3.6.1 Common case performance

In this section we evaluate Aardvark in the absence of failures. We compare the throughput and latency of Aardvark to select previous systems during gracious execution and evaluate the impact of the key differences between Aardvark and previous systems.

Failure-free performance. We first measure the throughput and latency of Aardvark and the competing systems in the absence of failures. The results are shown in Figure 3.6. We see that Aardvark's peak throughput is competitive with that of contemporary state of the art systems. Aardvark's throughput peaks at 38.7k operations per second, while Zyzzyva and PBFT observe maximum throughputs of 66k and 61.7k operations per second, respectively. The reliance on digital signatures to authenticate client requests increases the per request processing in Aardvark, resulting in increased per request latency and lower throughput. Aardvark, PBFT, and Zyzzyva provide higher throughput than HQ and Q/U because the former set of systems batch requests while the latter two systems process each request individually.

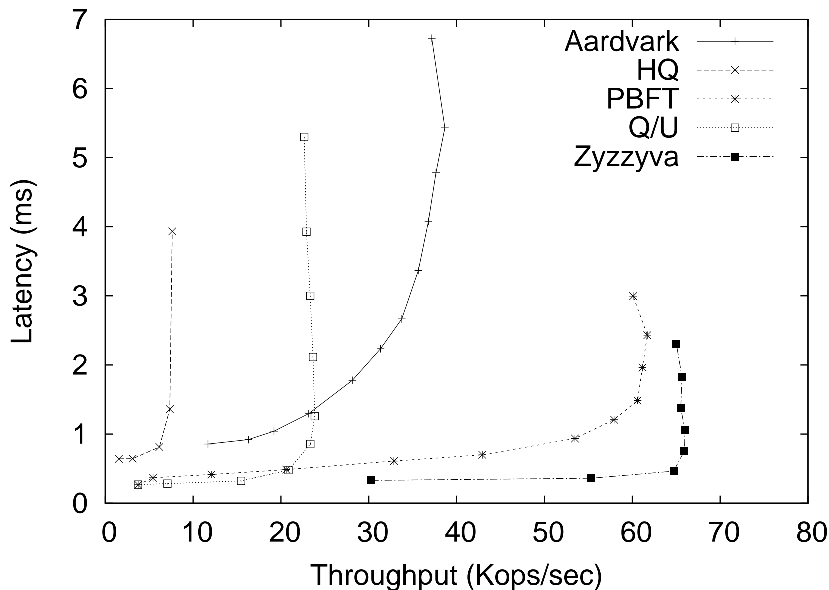

Figure 3.6: Average per request latency vs. average throughput for Aardvark, HQ, PBFT, Q/U, and Zyzzyva.

Putting Aardvark together. Aardvark incorporates several key design decisions that enable it to perform well in the presence of Byzantine failure. We study the performance impact of these decisions by measuring the throughput of several variants of PBFT and Aardvark. Each variation corresponds to a piece-wise evolutionary step from PBFT to Aardvark. We measure the peak throughput of each variant by increasing the offered workload until throughput stabilizes. We report the peak throughput of each variant in Table 3.2.

While requiring clients in PBFT to sign requests reduces throughput by 50%, we find that the cost of requiring Aardvark clients to use the hybrid MAC-signature scheme imposes a smaller 33% hit to system throughput. Explicitly separating the work queues for client and replica communication makes it easy for Aardvark to utilize the second processor in our test-bed machines, which reduces the throughput costs Aardvark pays to verify signed client requests. This parallelism is the primary source of the 30% improvement we observe between PBFT with signatures and Aardvark.

Peak throughput for Aardvark with and without regular view changes is

System	Peak Throughput
Aardvark	38.7k
PBFT	61.7k
PBFT w/ client signatures	31.8k
Aardvark without signatures	57.4k
Aardvark without regular view changes	39.8k

Table 3.2: Peak throughput of Aardvark and PBFT for different implementation choices.

comparable. The reason for this is rather straightforward: when both the new and old primary replicas are non-faulty, a view change requires approximately the same amount of work as a single instance of consensus. Aardvark views led by a non-faulty primary are sufficiently long that the throughput costs associated with performing a view change are negligible.

View Changes. We now explore the impact of performing regular view changes on the per request latencies observed by clients. We measure the latencies observed by 210 clients, each issuing 100k requests. Clients are configured to retransmit requests if they do not receive a response within 150ms of issuing the request.

Figure 3.7 shows the per-request latency observed by a single client during one run of the experiment. The periodic latency spikes correspond to view changes. When a client issues a request as the view change is initiated, the request is not processed until the request arrives at the new primary following a client timeout and retransmission. In most cases a single client retransmission is sufficient, but additional retransmissions may be required when multiple view changes occur in rapid succession.

Figure 3.8 shows the CDF for latencies of all client requests in the same experiment. We observe that a vast majority of requests have latency under 15ms[4] , and only a small fraction of all requests incur the higher latencies induced by view changes.

[4]Though it is not visible in the graph, we observe that 99.99% fo requests have throughput under 15ms

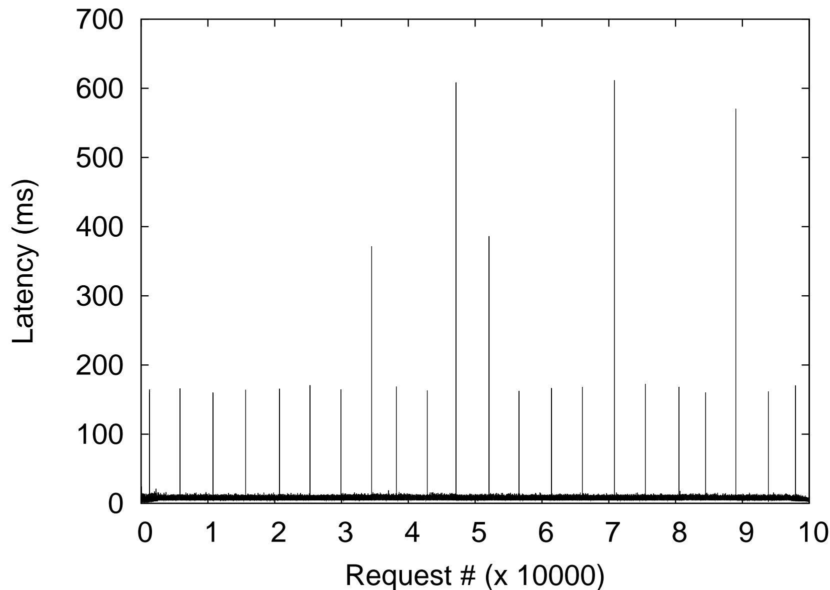

Figure 3.7: The latency of an individual client's requests running Aardvark with 210 total clients. The sporadic jumps represent view changes in the protocol.

3.6.2 Evaluating faulty systems

In this section we evaluate Aardvark and existing systems in the context of failures. It is impossible to test every possible Byzantine behavior; consequently we use our knowledge of the systems to construct a set of workloads that we believe to be close to the worst case for Aardvark and other systems. While other faulty behaviors are possible and may stress the evaluated systems in different ways, we believe that our results are indicative of both the vulnerability of existing systems and the robustness of Aardvark.

Faulty clients

We focus our attention on two aspects of client behavior that have significant impact on system throughput: request dissemination and network flooding.

Request dissemination. Table 3.1 (in Section 3.1) depicts the impact of faulty client behavior related to request distribution on the PBFT, HQ, Zyzzyva, and Aard-

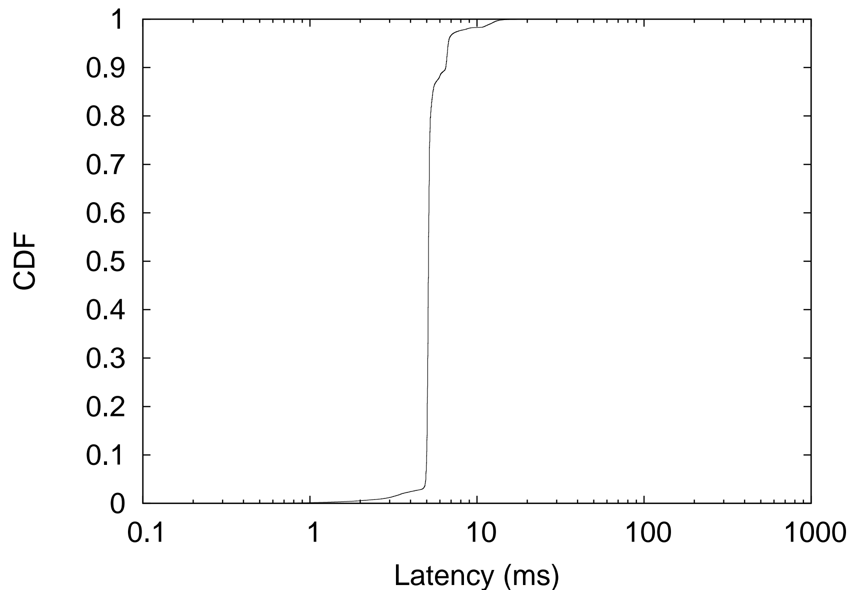

Figure 3.8: CDF of request latencies for 210 clients issuing 100,000 requests with Aardvark servers.

vark prototypes. We implement different client behaviors for the different systems to stress test the design decisions the systems have made.

In PBFT and Zyzzvya, the clients send requests that are authenticated with MAC authenticators. The faulty client includes an inconsistent authenticator on requests so that request verification will succeed at the primary but fail for all other replicas. When the primary includes the client request in a PRE-PREPARE message, the replicas are unable to verify the request.

We developed this workload because, on paper, the protocols specify what appears to be an expensive processing path to handle this contingency. In this situation PBFT specifies a view change while Zyzzyva invokes a conflict resolution procedure that blocks progress and requires replicas to generate signatures. In theory, these procedures should have a noticeable, though finite, impact on performance. In particular, PBFT progress should stall until a timeout forces a new view ([16] pp. 42–43), at which point other clients can make some progress until the faulty client stalls progress again. In Zyzzyva, the servers should pay extra overheads for signatures and view changes.

In practice, the throughput of both prototype implementations drops to 0. In Zyzzyva the reconciliation protocol is not fully implemented; in PBFT the client behavior results in repeated view changes, and we have not observed our experiment to finish. While the full PBFT and Zyzzyva protocol specifications guarantee liveness under eventual synchrony, the protocol steps required to handle these cases are evidently sufficiently complex to be difficult to implement, easy to overlook, or both.

In HQ, our intended attack is to have clients send certificates during the WRITE-2 phase of the protocol with an inconsistent MAC authenticator. The response specified by the protocol is a signed WRITE-2-REFUSED message which is subsequently used by the client to initiate a call to initiate a request processed by an internal PBFT protocol. This set of circumstances presents a point in the HQ design where a single client, either faulty or simply unlucky, can force the replicas to generate expensive signatures resulting in a degradation in system throughput. We are unable to evaluate the precise impact of this client behavior because the replica processing necessary to handle inconsistent MAC authenticators from clients is not implemented in the HQ prototype.

In Q/U during periods of contention when multiple clients issue concurrent requests that modify or depend on overlapping state, replicas are required to perform barrier and commit operations that are rate limited by a client-initiated exponential back-off. During the barrier and commit operations, a faulty client that sends inconsistent certificates to the replicas can theoretically complicate the process further. We implement a simpler scenario in which all clients are correct, yet they issue contending requests to the replicas. In this setting with only 20 clients, the throughput of the Q/U prototype also drops to zero. Q/U's focus on performance in the absence of both failures and contention makes it especially vulnerable in practice—clients that issue contending requests can decimate system throughput, whether the clients are faulty or not.

To avoid corner cases where different replicas make different judgments about the legitimacy of a request, Aardvark clients sign requests. In Aardvark, the closest client behaviors analogous to those discussed above for other systems are sending requests with a valid MAC and invalid signature or sending requests with invalid MACs. We implement both attacks and find the results to be comparable. In Table 3.1 we report the results for requests with invalid MACs. Aardvark does not

suffer from a throughput degradation comparable to the ones observed in previous systems because it is able to process the faulty requests efficiently. Requests with an invalid MAC are discarded quickly and do not induce any replica-to-replica communication. Similarly, requests with an invalid signature induce a high one time cost for the primary, but subsequent requests from that client are efficiently discarded. It is important to note that the client in this attack follows the retransmission schedule for a correct client. Our next discussion explores the impact of a clients and servers that aggressively transmit messages.

Network flooding. In Table 3.3 we demonstrate the impact of a single faulty client that floods the replicas with messages. During these experiments correct clients issue requests sufficient to saturate each system while a single faulty client implements a brute-force denial-of-service attack by repeatedly sending 9KB UDP messages to the replicas[5]. For PBFT and Zyzzyva, 210 clients are sufficient to saturate the servers, while Q/U and HQ are saturated with 30 client processes[6].

The PBFT and Zyzzyva prototypes suffer dramatic performance degradation as their incoming network resources are consumed by the flooding client; processing the incoming client requests disrupts the replica-to-replica communication necessary for the systems to make progress. In both cases, the pending client requests eventually overflow queues internal to the server processes resulting in a `seg-fault` and subsequent crash. Q/U and HQ suffer smaller degradations in throughput from the spamming replicas. The UDP traffic is dropped by the network stack with minimal processing because it does not contain valid TCP packets. The slowdowns observed in Q/U and HQ correspond to the displaced network bandwidth.

The reliance on TCP communication in Q/U and HQ changes rather than solves the challenge presented by a flooding client. For example, a single faulty client that repeatedly requests TCP connections crashes both the Q/U and HQ servers.

In each of these systems, the vulnerability to network flooding is a byproduct of the prototype implementation and is not fundamental to the protocol design. Network isolation techniques such as those described in Section 3.4 could similarly be applied to these systems.

[5]The faulty client is a modified PBFT client instrumented to repeatedly send messages of maximal size, 9KB in the release we evaluate.

[6]These client saturation numbers are specific to our experimental machines and closed-loop client construction

System	Peak Throughput	Network Flooding	
		UDP	TCP
PBFT	61.7k	crash	-
Q/U	23.8k	23.1k	crash
HQ	7.6k	4.5k	0
Zyzzyva	66k	crash	-
Aardvark	38.7k	7.8k	-

Table 3.3: Observed peak throughput of BFT systems in the fault free case and under heavy client retransmission load. UDP network flooding corresponds to a single faulty client sending 9KB messages. TCP network flooding corresponds to a single faulty client sending requests to open TCP connections and is shown for TCP based systems.

In the case of Aardvark, the decision to use separate NICs and work queues for client and replica requests ensures that a faulty client is unable to prevent replicas from processing requests that have already entered the system. The throughput observed by Aardvark tracks the fraction of requests that replicas receive that were sent by non-faulty clients.

Faulty Primary

In systems that rely on a primary, the primary controls the sequence of requests that are processed.

In Table 3.4 we show the impact on PBFT, Zyzzyva, and Aardvark prototypes of a primary that delays sending PRE-PREPARE messages by 1, 10, or 100 ms. The throughput of both PBFT and Zyzzyva degrades dramatically as the slow primary is not slow enough to trigger their view change conditions. This throughput degradation is a consequence of the protocol design and specification of when view changes should occur. With an extremely slow primary, Zyzzyva eventually succumbs to a memory leak exacerbated by holding on to requests for an extended period of time. The throughput achieved by Aardvark indicates that adaptively performing view changes in response to observed throughput is a good technique for ensuring performance.

In addition to controlling the rate at which requests are inserted into the system, the primary is also responsible for controlling which requests are inserted into the system. We evaluate this impact by instrumenting a single replica to defer

System	Peak Throughput	1 ms	10 ms	100 ms
PBFT	61.7k	5k	4.9k	1.1k
Zyzzyva	66k	27.8k	5k	crash
Aardvark	38.7k	38.5k	37.3k	37.9k

Table 3.4: Throughput during intervals in which the primary delays sending PRE-PREPARE message (or equivalent) by 1, 10, and 100 ms.

System	Starved Throughput	Normal Throughput
PBFT	1.25	1.5k
Zyzzyva	0	1.7k
Aardvark	358	465

Table 3.5: Average throughput for a starved client that is shunned by a faulty primary versus the average per-client throughput for any other client.

processing request from a specified client and report the throughput observed for the shunned client and average throughput for remaining clients. Table 3.5 depicts the results of this experiment. In the case of PBFT and Aardvark, the primary sends a PRE-PREPARE for the targeted client's request only after receiving the request 9 times. This heuristic prevents the PBFT primary from triggering a view change and demonstrates dramatic degradation in throughput for the targeted client in comparison to the other clients in the system. For Zyzzyva, the unfair primary ignores messages from the targeted client entirely. The resulting throughput is 0 because the implementation is incomplete, and replicas in the Zyzzyva prototype do not forward received requests to the primary as specified by the protocol. Aardvark's fairness detection and periodic view changes limit the impact of the unfair primary.

Non-Primary Replicas

We implement a faulty replica that does not process protocol messages and blasts network traffic at the other replicas instead. We report the results of running the systems with the blasting replica in Table 3.6. In the first experiments, a faulty replica blasts 9KB UDP messages at the other replicas[7]. The PBFT and Zyzzyva

[7] The PBFT prototype uses UDP for inter-server communication. The faulty client is implemented with a PBFT client that sends well-formed, but non-sensical, request messages.

System	Peak Throughput	Replica Flooding	
		UDP	TCP
PBFT	61.7k	251	-
Q/U	23.8k	19.3k	crash
HQ	7.6k	crash	crash
Zyzzyva	66k	0	-
Aardvark	38.7k	11.7k	-

Table 3.6: Observed peak throughput and observed throughput when one replica floods the network with messages. UDP flooding consists of a replica sending 9KB messages to other replicas rather than following the protocol. TCP flooding consists of a replica repeatedly attempting to open TCP connections on other replicas.

prototypes again show very low performance as the incoming traffic from the spamming replica displaces much of the legitimate traffic in the system, denying the system both requests from the clients and also replica messages required to make progress. Aardvark's use of separate worker queues ensures that the replicas process the messages necessary to make progress. In the second experiment, the faulty Q/U and HQ replicas again open TCP connections, consuming all of the incoming connections on the other replicas and denying the clients access to the service.

Once again, the shortcomings of the systems are a byproduct of implementation and not protocol design. We speculate that improved network isolation techniques would make the systems more robust.

3.7 Conclusion

We claim that high-assurance systems require BFT protocols that are more robust to failures than existing systems. Specifically, BFT protocols suitable for high-assurance systems must provide adequate throughput during uncivil intervals in which the network is well behaved but an unknown number of clients and up to f servers are faulty. We present Aardvark, the first BFT state machine protocol designed and implemented to provide good performance in the presence of Byzantine faults. Aardvark sacrifices peak throughput during gracious executions in order to gain significant improvement in performance during uncivil executions.

This chapter contains two important contributions. The first contribution is the design, presentation, and evaluation of the Aardvark prototype. The second,

and most important, contribution is the observation that fault tolerant systems should be robust to failures—it is not enough to ensure safety and eventual liveness if, during a synchronous interval, a faulty node can reduce system throughput to unacceptably low levels. This simple observation is most notable for its absence in the discussion of previous systems [1, 18, 26, 45, 50, 49, 67, 86, 92, 104, 107]. While the discussion in this chapter has focused on the specific example of asynchronous BFT RSM protocols, we believe that robust performance is an important goal for *all* fault tolerant systems.

Chapter 4

UpRight RSM Architecture

State machine replication is a powerful technique for building reliable services from faulty components [52, 88]. The basic approach is simple: convert an application to a deterministic state machine, replicate the state machine, and ensure that each replica executes the same set of requests in the same order. Clients then gather votes from multiple replicas to determine the correct response to deliver to the user. The ultimate goal of an application built on top of an RSM protocol is to ensure that the set of responses received by users of the replicated service are indistinguishable from a set of responses that could have been generated by a single correct server given the same set of requests.

Indeed, the systems described in the previous chapter (and many others [1, 12, 18, 24, 26, 49, 50, 67, 92, 104, 108]) are based on state machine replication. These libraries typically specify a linearized order of requests and require replicated applications to execute the requests in the specified linearized order. Initial work towards accommodating parallel execution has focused on requiring the application to execute requests so that the responses and resulting state are equivalent to executing the requests in the specified order [50].

The rest of this thesis focuses on the design, implementation, and use of the UpRight library for state machine replication. The UpRight library is a new library for state machine replication that differs from previous fault tolerant RSM libraries in three important ways.

First, the UpRight library is the first RSM library designed to provide (a) UpRight fault tolerance (as opposed to the customary crash[12, 53, 67, 108] or

Byzantine [18, 24, 26, 49, 50, 92, 104] fault tolerance) and (b) robust performance from the outset. While statements of fault tolerance equivalent to UpRight fault tolerance have been described before [1, 32, 55], these statements have not resulted in systems designed to leverage the flexibility and low replication costs of UpRight fault tolerance. Similarly, although the Aardvark prototype discussed in Chapter 3 instantiates RBFT, it is accurate to say that robustness was shoe-horned onto an existing library (PBFT [18]). Starting the design with robustness and UpRight tolerance as initial goals allows for a clean end-to-end design. Chapters 2 and 3 provide the foundation for understanding UpRight and robust fault tolerance.

Second, the architecture of the UpRight library is based on three distinct stages corresponding to the key steps of state machine replication: request authentication, agreement on an execution order, and deterministic execution of the specified order[1]. The latter two stages are standard and form the core of Schneider's definition of state machine replication [89]. The distinction between these two stages has been leveraged in previous system designs [60, 104, 107] in order to reduce the required number of replicas[2]. The request authentication stage is a new stage introduced for the UpRight library as a direct result of our experience the Aardvark prototype and robust fault tolerance (Chapter 3). The three stages of the UpRight architecture have distinct replication requirements, and designing the library around the three stages facilitates efficient usage of computer resources in deployed systems.

Third, the UpRight library refines the responsibilities and expectations of the library and replicated applications. The refined responsibilities are reflected in two key differences between API exposed by the UpRight library and the API of previous libraries. First, the UpRight library requires a replicated application to deterministically execute *batches* of requests in a linearized order; in contrast, previous libraries [18, 24, 49, 50, 92, 104, 107] have required replicated applications to deterministically execute a linearized order of requests. This change in semantics exposes the efficiency-driven internal batching performed by libraries [18, 24, 49, 50, 92, 104, 107] to the application and explicitly emphasizes the possibility of executing non-conflicting requests within the batch in parallel, providing a solid basis for

[1]The client side of the library can be considered a fourth stage.

[2]It is important to note that most existing systems combine agreement and execution into a single stage [1, 18, 24, 26, 49, 50, 53, 92].

leveraging parallel hardware and resources. Second, the UpRight library requires a replicated application to deterministically produce checkpoints on demand; previous replication libraries have taken the checkpoint for the application. While generating deterministic checkpoints on demand initially seems like an extra burden for the application programmer, we believe that it is actually simpler than current techniques requiring the application to be rewritten to support a memory model defined by the replication library.

The rest of this chapter is organized as follows. Section 4.1 provides an overview of the UpRight architecture. Section 4.2 describes the responsibilities of the UpRight library and replicated applications in more detail. Section 4.3 previews the subsequent chapters and explains the relationship between those chapters to the ideas discussed in this chapter.

4.1 UpRight architecture

There are a multitude of design decisions that go into building systems. Many of these decisions, i.e. using MACs rather than digital signatures to authenticate messages, appear straightforward; it is easy to hope that they can be introduced through local pinhole changes. The reality is that these "small" design decisions can have wide-reaching impact on the end-to-end system. For example, the decision to replace digital signatures with MACs improves performance (i.e. reduces latency and increases throughput) in the common case, but introduces an expensive corner case that can lead to significant performance degradation as shown in Section 3.6.2.

We design the UpRight library to replicate the servers, using state machine replication, in client-server systems. The "client" portion of the system consists of the client-side application (aka *user*) and a library client. The "server" portion of the system consists of the original application server and the library components used to coordinate multiple replicas of the application server.

Figure 4.1 provides a graphical depiction of a client-server system implemented with the UpRight library. The client portion of the UpRight library is responsible for interfacing between the application-level user code and the replicated server. The server portion of the UpRight library consists of three distinct stages: (1) request authentication stage, (2) request ordering stage, and (3) request execution stage. Each stage fulfills a specific function: the authentication stage in-

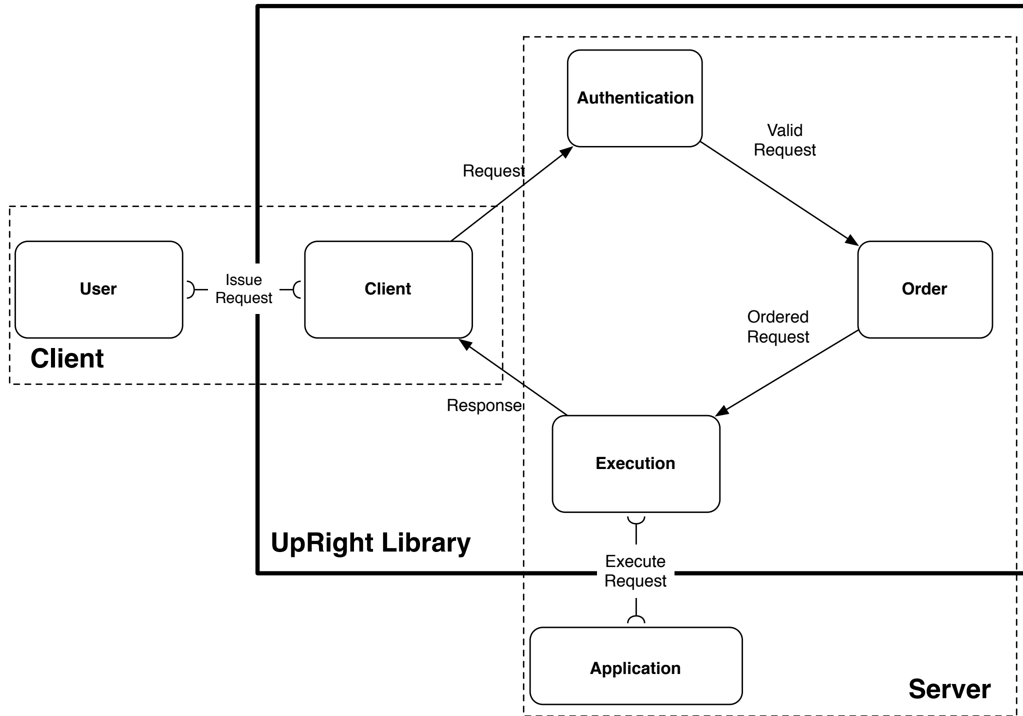

Figure 4.1: Basic flow of messages in the UpRight architecture.

sures that client requests are valid, the order stage places valid requests in batches and orders the batches, and the execution stage delivers batches to the application and relays application responses to clients.

Separating the server side of the UpRight library into three distinct stages allows us to provide clean solutions to problems that arise as we replicate the server and also to address shortcomings in previous replicated systems. Separating the authentication stage allows us to (a) authenticate client requests at low cost in both the average and worst cases (i.e. avoid the dangers of faulty clients (Section 3.6.2) without relying on public key cryptography) and (b) minimize the overall network bandwidth and costs associated with ordering requests. Separating order and execution allows us to (a) reduce the overheads of ordering and (b) reduce the total computation in the system by replicating each stage the minimum amount required for that stage rather than the maximum replication required for any stage[3].

[3]The benefits of separating order from execution have been noted by others [60, 104, 107]. We

At a high level, the protocol is simple: clients send requests to the authentication stage; the authentication stage authenticates the requests; the order stage assigns each authenticated request to a batch and assigns an execution order to the batches; the execution stage executes the batches of requests (by delivering them to the application) in the specified order and reports the results back to the clients.

Of course, the reality is more complex than this high level view implies, as various factors conspire to complicate the design—individual nodes can fail in unpredictable ways, the network may not be reliable, and computational/storage/network resources are finite. We will discuss the interaction between stages and challenges associated with replicating each stage for reliability in subsequent chapters.

This description also glosses over the interactions between the library and the application. We explore that set of interactions in the next section with a specific focus on the properties that the library and application are required to uphold.

4.2 Division of responsibilities

The previous section describes the internal architecture of the UpRight library. In this section, we focus on the contract between the UpRight library and replicated applications. Section 4.2.1 details the responsibilities of the UpRight library. Section 4.2.2 details the responsibilities of replicated applications.

4.2.1 Library properties

The UpRight library delivers a linearized sequence of batches of one or more requests to the application. In addition to guaranteeing that each application replica receives the same sequence of request batches, the UpRight library ensures that only authorized requests are included in the ordered batches and that the batches themselves are well-formed.

The key difference between the properties of the UpRight library and previous libraries is that the UpRight library defines a linearized order for *batch* execution while previous libraries define a linearized order for *request* execution. In otherwords, the Upright library defines a partial order for request execution rather than the total order defined by previous systems.

take care to address technical challenges related to checkpoint coordination overlooked in previous efforts to leverage that separation.

Before specifying the properties provided by the UpRight library, we first describe some basic notation. A batch of requests is identified by an identifier n_o. A batch contains a set of one or more requests each issued by an authorized client c, an associated pseudo-random number generator (PRNG) seed, and an associated time t. Requests from client c are differentiated by a request identifier n_c and each request is placed in at most one batch. Batch n_o is well-formed if it contains at most one request per authorized client c.

The UpRight library provides the following safety properties:

LS1 Only responses generated by the application are delivered to non-faulty users.

LS2 Only non-empty batches are delivered to the application.

LS3 Batch n_o is only delivered to the application if the previously delivered batch is $n_o - 1$.

LS4 Request n_c issued via client c is placed in at most one batch.

LS5 If requests n_c and n'_c issued via client c are in batches n_o and n'_o respectively, then $n_c < n'_c$ iff $n_o < n'_o$.

LS6 Only requests issued to an authorized client are placed in a batch[4].

LS7 Given batches n_o and n'_o and associated times t and t', $n_o < n'_o \rightarrow t < t'$.

The UpRight library provides two distinct liveness properties. First, any request issued by an authorized (and correct) client is delivered to the application. Second, any response generated by the application is delivered to the user that issued the request.

LL1 Any request issued via a correct client is eventually delivered to the application.

LL2 Any application-produced response to a request issued via correct client c is eventually provided to c and delivered to the user.

[4]A client is "authorized" if it has appropriate credentials to interact with the servers. It is the responsibility of the sysadmin to secure access to authorized clients.

4.2.2 Application requirements

We explicitly depart from the "standard" requirements imposed on applications by replication libraries in the PBFT lineage [18, 86, 24, 26, 49, 92, 107, 104] in two important ways. First, as stated above, we require applications to execute batches, rather than individual requests, in a specified linearized order. Second, we explicitly charge the application with taking and loading checkpoints. Previous libraries have provided automatic checkpointing functionality and required the application to place all relevant information in a library managed (and checkpointed) memory space.

We define some basic terminology that is useful in understanding the application properties and API. Let $\mathcal{H}$ be a linearized sequence of ordered batches. Let $\mathcal{H}^i$ be the sequence of the first i ordered batches. Let $\mathcal{S}_{\mathcal{H}}$ be the state of the application after executing every batch in $\mathcal{H}$ in order. Let $\mathcal{C}_{\mathcal{H}}$ be the checkpoint of state $\mathcal{S}_{\mathcal{H}}$. Let $\mathcal{H} : b$ be a linearized sequence of ordered batches where b is the last batch in the sequence. Let $\mathcal{R}_b$ be a set of responses generated when processing batch b.

We require the application to implement three basic functions: execute a batch, take a checkpoint, and load a checkpoint. The execute batch function $exec : \mathcal{S} \times b \to \langle \mathcal{S}, \mathcal{R} \rangle$ is a function from an application state and a batch of requests that transitions the application to a new state and produces a set with one response per request in the batch. The take checkpoint function $takeCP : \mathcal{S} \to \mathcal{C}$ generates a checkpoint that describes a valid application state and the load checkpoint function $loadCP : \mathcal{S} \times \mathcal{C} \to \mathcal{S}$ sets the application state to the state described by the specified checkpoint[5][6].

In a break from our normal pattern, we specify the application liveness properties first:

APPL1 $exec(\mathcal{S}, b)$ returns a set of responses $\mathcal{R}_b$.

APPL2 $takeCP(\mathcal{S})$ returns a checkpoint $\mathcal{C}$.

APPL3 $loadCP(\mathcal{S}, \mathcal{C})$ returns.

[5] Note that *exec*, *takeCP*, and *loadCP* are not functions in the strictly mathematical sense. They correspond instead to functions in a programming API, as such there is no guarantee that *exec* will produce the same output each time it is provided with a specified input.

[6] Note that *exec* and *loadCP* don't return an application state $\mathcal{S}$, but rather transition the application to be in the specified state

In short, the liveness properties correspond to providing a terminating implementation of the execute, take checkpoint, and load checkpoint functions.

The required application safety properties are more interesting as they aggressively restrict the behavior of the application:

APPS1 Only requests contained in batches received from the library are executed.

APPS2 If $exec(\mathcal{S}_{\mathcal{H}}, b) = \langle \mathcal{S}_{\mathcal{H}:b}, \mathcal{R}_b \rangle$ and $exec(\mathcal{S}_{\mathcal{H}}, b) = \langle \mathcal{S}'_{\mathcal{H}:b}, \mathcal{R}'_b \rangle$ then $\mathcal{S}_{\mathcal{H}:b} = \mathcal{S}'_{\mathcal{H}:b}$ and $\mathcal{R}_b = \mathcal{R}'_b$

APPS3 If $takeCP(\mathcal{S}_{\mathcal{H}}) = \mathcal{C}_{\mathcal{H}}$ and $takeCP(\mathcal{S}_{\mathcal{H}}) = \mathcal{C}'_{\mathcal{H}}$ then $\mathcal{C}_{\mathcal{H}} = \mathcal{C}'_{\mathcal{H}}$

APPS4 $\forall \mathcal{S} : loadCP(\mathcal{S}, takeCP(\mathcal{S}_{\mathcal{H}})) = \mathcal{S}_{\mathcal{H}}$

APPS1 ensures that the application does not execute random requests. APPS2 ensures that the application executes batches deterministically. Deterministic batch execution is (a) useful when running multiple copies of the application and (b) vital because the UpRight library relies on execution replay to tolerate transient crashes. Note that deterministic batch execution requires that the application, when given the same sequence of batches from a specified starting state, (a) reaches the same final application state and (b) generates the same set of responses. APPS3 ensures that checkpoints generated by the application are deterministic based on when in the execution sequence take checkpoint is called. APPS4 ensures that after loading a checkpoint, the application is in the same state as when take checkpoint was called.

While these properties are intuitive, they are not intrinsic to every application. APPS2 for example, is violated by any application that puts a system time-stamp on every response it generates as rolling back the application state and re-executing requests would result in a different set of time-stamps on the generated responses. APPS3-4 are violated by the checkpoints generated by replicas in the ZooKeeper distributed coordination service [108]. ZooKeeper replicas generate checkpoints at time t by asynchronously recording an application snapshot to stable storage and logging all requests processed between time t and when the snapshot is completed. The resulting checkpoint is loaded through a two-step process of first loading the snapshot and then replaying the log of requests[7]. This checkpoint procedure ensures a pair of weaker properties than we target. Specifically, ZooKeeper ensures that

[7] ZooKeeper requests are idempotent.

$takeCP(\mathcal{S}_{\mathcal{H}}) = \mathcal{C}_{\mathcal{K}}$ and $\forall S' : loadCP(\mathcal{S}', takeCP(\mathcal{S}_{\mathcal{H}})) = \mathcal{S}_{\mathcal{J}}$ where $\mathcal{H}$ is a prefix of both $\mathcal{K}$ and $\mathcal{J}$.

4.3 Looking forward

This chapter lays the framework for the design of the UpRight library and the expectations we place on the library and replicated applications. The chapters that follow expand on this key points made in this chapter and the interplay between the goals laid out in this chapter and the design and use of the UpRight library.

Chapter 5 describes the stage level architecture in detail. We focus the discussion on the interaction between correct stages and the properties that each stage must uphold.

Chapter 6 describes the replication of each stage. We focus the discussion on how we provide replicated instantiations of the stages that fulfill the properties defined in Chapter 5 in a robust UpRight fault tolerant manner.

Chapter 7 describes our experiences with modifying the ZooKeeper distributed coordination service [108] and Hadoop Distributed File System [43] to be compatible with the UpRight library. Our primary interest in Chapter 7 is evaluating the complexity of adapting an existing application to be compatible with the UpRight library.

Future work remains on expanding on the techniques employed by Kotla et al. [50] to general applications or developing novel approaches to achieving deterministic parallel execution.

Chapter 5

UpRight Stages

This chapter treats each *stage* of the UpRight architecture as a unit, ignoring internal replication details. In practice each stage is implemented by a set of nodes, but to the extent possible we abstract away that detail in this chapter. In particular, the replication we discuss in Chapter 6 will mask individual node crash, omission, and commission failures in the three stages and allows us to treat the stages as abstract correct entities in the current discussion. Our goal in this chapter is to fully describe the messages exchanged between stages[1] and the properties that each stage provides.

In the context of this chapter we differentiate between *correct* and *idealized* stage. An idealized stage follows its specification faithfully and is not limited by practical constraints such as limited memory or power outages. In contrast, a correct node follows its specification faithfully, but has limited memory and is subject to temporary power outages. Even though the replication within each stage masks failures of individual nodes and provides the abstraction of a correct stage, there are multiple challenges that we must address in the stage-level design: (1) clients can be faulty, (2) the network may not be reliable, (3) network resources are limited, (4) node (and by extension stage) storage resources are limited, and (5) an entire stage may transiently crash (i.e. temporary power failure).

First, clients can fail. While we can rely on replication to ensure that a stage is correct, clients cannot be replicated and many systems would not trust

[1]When discussing the stage-level protocol we refer to a single message. The stage replication discussed in Chapter 6 requires most messages to be sent to every node in the next stage. Receiving messages from a replicated stage generally requires the recipient to gather a quorum of matching messages.

clients to be correct even if client replication were possible. There are two distinct challenges associated with handling client failures: we must ensure that requests issued by correct clients are processed by the system and we must ensure that faulty clients are unable to corrupt the system state or prevent correct clients' requests from being processed. Our high-level solution is to specify a contract for clients to follow—requests issued by a client that fulfills its part of the contract are eventually executed while no guarantees are provided to clients that violate the contract.

Second, lossy and asynchronous networks can lose and arbitrarily delay messages. Messages can be lost between any pair of stages: a client request may not reach the authentication stage, an authenticated request may not reach a order stage, an ordered request may not reach the execution stage, and a result may not reach the client. Thus, our second challenge is to ensure that the system is safe and live despite the loss or delayed delivery of various messages. It is well known that it is impossible to achieve safe and live operation in the presence of an asynchronous network with failures [35]; we consequently target liveness only during sufficiently long synchronous intervals. Our high-level solution is to ensure that each stage caches relevant messages in transient memory for retransmission as needed. Note that retransmission can be caused by a *push*, i.e., the client retransmits a request if it does not receive a timely response, or a *pull*, i.e., the execution stage requests the retransmission of ordered batch i if it receives batch $i+1$ first.

Third, network bandwidth is a limited resource. There is a limit to the number of bytes that can be exchanged between nodes. This limit becomes especially important when we consider one of the requirements of the replicated order stage: in order to order a request, every order node must have a copy of the request. Thus, our third challenge is to be economical in our use of network resources and avoid sending unnecessary information between stages or the nodes ins a given stage. Our high-level solution is to order cryptographic hashes of requests hashes and store the request bodies at the authentication stage. The key observation is that request bodies can skip the order stage entirely and go straight to the execution stage.

Fourth, nodes have finite memory. The nodes in each stage cannot maintain the arbitrary number of messages that they may be required to retransmit for messages that are lost in transmission. Thus, our fourth challenge is to ensure that each stage *garbage collects* messages without impeding the system's ability to make safe and consistent progress (e.g., by discarding a message that may still be needed by

another stage). Our high-level solution is to take order and execution checkpoints at pre-determined intervals and garbage collect messages that are made obsolete by recent checkpoints. Efficient checkpoint generation and garbage collection requires us to pay careful attention to the state stored, the timing of when checkpoints are taken, and when stored state can be garbage collected at each stage.

Fifth, even though replication can mask failure of some subsets of nodes (i.e. ensure liveness despite u failures and safety despite r commission failures), the nodes in a stage can transiently crash at arbitrary times (e.g. due to power failure). When a node crashes it loses the contents of transient memory, but storing data in persistent memory that survives transient failures is an expensive operation that we would like to avoid when possible. Avoiding frequent access to persistent memory is especially important at the execution stage, which is colocated with an application that may rely heavily on persistent memory access as part of processing requests. Thus, our final challenge is to ensure that sufficient state is stored in persistent memory to allow the system to efficiently resume operation following a transient crash of individual nodes or all nodes in a stage while at the same time limiting the extent to which persistent memory becomes a performance bottleneck. Our high-level solution is to store in persistent memory (1) every request sent by the authentication and order stages and (2) order and execution checkpoints. This solution provides sufficient causal logging [5] to ensure that no information is lost due to catastrophic power failure while minimizing persistent memory contention at the execution stage.

The rest of this chapter is organized as follows. Section 5.1 presents a simplified end-to-end protocol across stages targeted at asynchronous networks and correct stages. In the context of this discussion we address the challenges associated with faulty clients and an unreliable asynchronous network; we do not consider the limitations of finite resources or transient crashes. Section 5.2 describes our basic approach for handling network bandwidth limitations. Section 5.3 describes our approach to handling the challenges of finite memory resources and transient crashes. Section 5.4 compiles the full set of safety and liveness properties for each stage into a collection of tables for easy reference. Section 5.5 describes performance optimizations supported by the UpRight prototype. Section 5.6 establishes notational conventions that we rely on in Chapter 6 and Appendix A. Section 5.7 contains extensive pseudo-code and description for each stage.

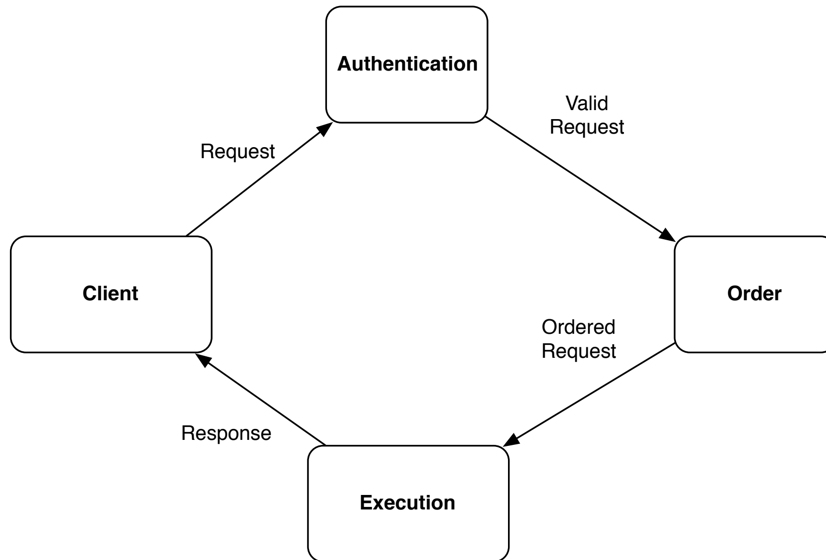

Figure 5.1: Message flow between idealized stages in the UpRight architecture.

5.1 Basic stage interactions

We present a simplified version of the interactions between stages intended to provide a solid intuition for our goals at each stage and for how the stages interact with clients and each other. Our initial description focuses on the basic properties provided by each stage and on how the stages combine to provide the end-to-end properties defined in Section 4.2.1. In this initial description, we describe the interactions between idealized stages that have infinite memory, are not subject to transient failures, and always follow their specification faithfully; we allow for an unreliable asynchronous network and do not assume that the network or clients are correct. The communication between idealized stages is shown in Figure 5.1.

We address the challenges of a faulty network and faulty clients in the expected ways. We rely on clients to retransmit requests until they receive a response, we ensure that all executed requests are executed safely, but we only promise that requests issued by correct clients will be executed. We do not promise anything to faulty clients.

5.1.1 Client properties

Clients issue requests and accept responses to those requests. As clients cannot be assumed correct, we specify the expected behavior of correct clients—i.e. their side of the contract. A correct client c upholds a pair of safety properties:

CS1 Each request issued by client c is assigned a unique request identifier n_c starting with 1 and increasing with each subsequent request.

CS2 Client c operates in a closed loop: it does not issue request $n_c > 1$ unless it has received a response to request $n_c - 1$.

Clients also uphold a single liveness property:

CL1 Client c resends request n_c until it receives a response.

Note that we explicitly consider requests to be different if they are issued by different clients or issued by the same client but with different request identifiers. Additionally, we implicitly assume that requests issued by client c depend on each other: $n_c + 1$ depends on n_c and so forth.

Clients view the service as a black box. If client c upholds CS1-2 and CL1 then it eventually receives a response to every request that it issues. If c fails to uphold any of the properties then it may or may not receive a response.

5.1.2 Authentication properties

The authentication stage validates client requests and sends authenticated copies of those requests to the order stage.

The authentication stage validates a request OP with request identifier n_c from client c if (a) the request is verifiably issued by client c, (b) it has not received any request $n'_c > n_c$ from client c, and (d) it has not received a request $\text{OP}' \neq \text{OP}$ from client c with request identifier n_c. We say that a request is *authenticated* when the authentication stage creates an authenticated request message containing the request and its associated authentication credentials. A correct verifier[2] a that receives an authenticated request message directly from the authentication stage can verify the authenticity of the request based on the authentication credentials.

[2]A *verifier* is any node tasked with verifying the authenticity of a received message. In this context, it refers to a node that receives a message from the authentication stage.

We say that an authenticated request message m is *one-step transferable* if, when a provides m received from the authentication stage to verifier b both a and b make the same determination about the authenticity of the request. Note that if b subsequently passes m to a third verifier c, one-step transferability says nothing about the consistency between the conclusions drawn by verifiers b and c.

- AS1 Only requests issued by authorized clients are authenticated and every authenticated request is one-step transferable.

The authentication stage also provides a simple liveness property:

- AL1 If the authentication stage receives a request n_c issued by correct client c, then an authenticated request message containing request $n'_c \geq n_c$ is sent to the order stage.

AL1 implies a potentially unexpected handling of retransmitted client requests: if the authentication stage has authenticated request n_c from client c and subsequently receives request $n'_c < n_c$ from c, then it resends authenticated request n_c to the order stage. This behavior converts "old" requests to the most recent request processed for a specific client and does not impact correct clients that issue requests in a closed loop. The one-step transferable component of AS1 simplifies the design of the replicated order stage by circumventing challenges associated with handling the Big MAC attack discussed in Chapter 3.

5.1.3 Order properties

The order stage receives valid requests from the authentication stage, places them into a batch, and assigns an execution order to the batches. The order stage places each batch into a next-batch message which is sent to the execution stage. The order stage provides the following safety properties:

- OS1 Only client requests authenticated by the authentication stage are placed into batches, and request n_c issued by client c is placed in at most one batch.

- OS2 Batches contain one or more requests and are assigned monotonically increasing batch identifiers n_o starting with 1 and increasing by 1 with each subsequent batch. For batches n_o and n'_o with associated times t and t', $n_o > n'_o \rightarrow t > t'$.

OS3 If request $n_c > 1$ issued by client c is in batch n_o, then request $n_c - 1$ issued by client c is in batch $n'_o < n_o$.

Note that the order stage enforces CS2; the order stage orders request n_c for client c only if request $n_c - 1$ has already been ordered. Hence, requests from a faulty client that does not uphold CS2 are not processed.

The liveness properties ensured by the order stage are straightforward:

OL1 If the order stage receives unordered authenticated request n_c issued by correct client c, then the order stage places the request in batch n_o and eventually sends a next-batch message containing n_o to the execution stage.

OL2 If the order stage receives authenticated request n_c from client c that is already in batch n_o, then it instructs the execution stage to retransmit a response to request n'_c from client c in batch n'_o where $n'_c \geq n_c$ and $n'_o \geq n_o$ by sending a retransmission message.

OL3 If the execution stage requests all batches after n_e and the order stage has ordered batches through $n_o > n_e$, then the order stage resends all ordered batches from $n_e + 1$ through n_o inclusive.

During normal operation, the order stage receives each authenticated client request once, places the request in a batch, and sends the batch to the execution stage for execution.

An unreliable asynchronous network can drop messages arbitrarily; when that occurs, some form of retransmission is necessary. Dropped messages and retransmissions impact the order stage in two ways.

First, if client c does not receive a response to request n_c, then it retransmits the request until it receives a response. This retransmission, in conjunction with the authentication stage, can result in the order stage receiving authenticated request n_c multiple times. When the order stage receives from c a request n_c that has already been ordered, it instructs the execution stage to retransmit the response to that request—if the order stage has ordered a subsequent request $n'_c > n_c$ from c then it requests retransmission of the response to request n'_c instead. Note that correct clients operate in a closed loop and are consequently not impacted by the retransmission of a later request.

Second, if the execution stage misses an ordered batch (i.e. receives batch n_o but not batch $n_o - 1$) then it requests the missing batches from the order stage. The order stage responds to the execution stage with the collection of missing batches.

5.1.4 Execution properties

The execution stage delivers batches to the application in the order specified by the order stage. Each batch is delivered to the application exactly once, and the responses provided by the application after executing the contained requests are cached by the execution stage for potential future retransmission.

We say that the execution stage is in state n_o if it has executed every batch with batch identifier at most n_o. The execution stage transitions from state n_o to state $n_o + 1$ when it executes batch $n_o + 1$.

We assume that the application executes a batch of requests instantaneously. This assumption simplifies the discussion by masking additional complexity that can be accounted for through engineering the execution stage and application. It does not change the conceptual properties or relationship between the execution stage and the rest of the system.

The execution stage provides three safety properties.

ES1 Batch n_o is delivered to the application only if the last batch delivered to the application is $n_o - 1$.

ES2 Only ordered batches are delivered to the application

ES3 Only responses generated by the application are cached or sent to clients.

We note that ES1 and ES2 are closely related but distinct properties. ES1 ensures that ordered batches are delivered to the application in the specified order. ES2 ensures that *only* ordered batches are delivered to the application, ES2 specifically prevents arbitrary requests that are not included in an ordered batch from being delivered to the application.

Intuitively, the execution stage ensures that every ordered batch is executed. To achieve that goal, we rely on the following liveness properties:

EL1 If the execution stage receives ordered batch n_o and the last batch it delivered to the application is $n_o - 1$, then the execution stage delivers batch n_o to the application.

EL2 If the execution stage receives a response from the application, then it stores the response for retransmission and sends the response to the client that issued the corresponding request.

EL3 If the execution stage receives a retransmission instruction for request n_c from c in batch n_o and the last batch executed by the execution stage is $n_e > n_o$, then the execution stage resends the response to the most recent request $n'_c \geq n_c$ executed for client c.

EL4 If the execution stage receives a retransmission instruction for request n_c from client c in batch n_o and the last batch executed by the execution stage is $n_e < n_o$, then the execution stage informs the order stage that it has missed the batches since n_e.

Note that the execution stage notifies the order stage that it has missed a collection of ordered batches only after receipt of a retransmission request and *not* when it receives ordered batch messages out of order. This counter-intuitive decision is driven by how we handle limited resources and will be discussed in more detail in Section 5.3.

5.1.5 Putting the stages together

Given the stage properties identified above, demonstrating that the combination of stages maintains the desired end-to-end properties is straightforward. Recall the properties that the library is expected to uphold as defined in Section 4.2.1:

LS1 Only responses generated by the application are delivered to non-faulty users.

LS2 Only non-empty batches are delivered to the application.

LS3 Batch n_o is only delivered to the application if the previously delivered batch is $n_o - 1$.

LS4 Request n_c issued via client c is placed in at most one batch.

LS5 If requests n_c and n'_c issued via client c are in batches n_o and n'_o respectively, then $n_c < n'_c$ iff $n_o < n'_o$.

LS6 Only requests issued to an authorized client are placed in a batch[3].

LS7 Given batches n_o and n'_o and associated times t and t', $n_o < n'_o \rightarrow t < t'$.

LL1 Any request issued via a correct client is eventually delivered to the application.

LL2 Any application-produced response to a request issued via correct client c is eventually provided to c and delivered to the user.

Note that the two liveness properties apply only to requests issued via *correct* clients.

Theorem 4. *If the authentication, order, and execution stages uphold their respective safety properties then* LS1-7 *hold.*

Proof. LS1: Follows from ES3.
LS2: Follows from ES2.
LS3: Follows from ES1.
LS4: Follows from OS1.
LS5: Follows from OS3.
LS6: Follows from OS1 and AS1.
LS7: Follows from OS2.

□

Lemma 1. *Given eventual synchrony and correct authentication, order, and execution stages* LL1 *holds.*

Proof. It follows from CL1 that correct client c issues request n_c until it receives a response. There are two cases to consider: (1) the client receives a response, (2) the client does not receive a response.

Case 1: The client c has received a response. It follows from ES3 that only responses generated by the application are sent to c. It follows from APPS1 that only requests in batches delivered to the application by the execution stage are executed. □.

[3]A client is "authorized" if it has appropriate credentials to interact with the servers. It is the responsibility of the sysadmin to secure access to authorized clients.

Case 2: The client has not received a response. It follows from CL1 that the client will issue request n_c arbitrarily often. It follows from eventual synchrony that the request is received by the authentication stage arbitrarily often.

It follows from AL1 that some request $n'_c \geq n_c$ from c is authenticated arbitrarily often and from CS2 that that request is n_c. It follows from eventual synchrony that the request is received by the order stage arbitrarily often.

Request n_c will be placed in a batch and a retransmission instruction for request n_c is sent to the execution stage arbitrarily often. The first time request n_c is received by the order stage, it follows from CS1 that OS1 is satisfied. It follows from OL1 that n_c is ordered in batch n_o and batch n_o is sent to the execution stage. Every subsequent time that request n_c is received it follows from OL2 that a retransmission instruction for request $n'_c \geq n_c$ is sent to the execution stage. It follows from CS2 that $n'_c = n_c$. It follows from eventual synchrony that the retransmission instruction is received by the order stage arbitrarily often.

Upon receipt of a retransmission instruction by the execution stage there are two possibilities to consider: (1) the batch containing the request has been delivered to the application or (2) the batch has not been delivered. In the first scenario we are done. Consider the second scenario. It follows from EL4 that the execution stage sends the last executed notification to the order stage arbitrarily often. It then follows from OL3 that the order stage sends the missing batches until they are no longer missing or arbitrarily often. It follows from eventual synchrony and induction that all batch messages will eventually be received by the execution stage and that the batches are subsequently delivered to the application. □

Lemma 2. *Given eventual synchrony and correct authentication, order, and execution stages* LL2 *holds.*

Proof. It follows from EL1 that the response is cached and sent to the client. If it is received, then we are done.

Otherwise, it follows from CL1 that a correct client reissues its request until it receives a response. It follows from AL1 that the request is authenticated and sent to the order stage. It follows from APPS1 that only requests contained in batches are executed and from ES2 that only ordered batches are delivered to the application. It then follows that n_c has already been ordered so by OL2 a retransmission message is sent. It then follows from EL4 that the response is retransmitted to the client. The

above happens arbitrarily often and by eventual synchrony the response is eventually received by the client.

□

5.2 Network efficiency

When network resources are limited it is important to limit the number of bytes sent across the network. Our basic description of the stage-level protocol has the authentication stage sending authenticated requests to the order stage and the order stage sending those requests to the execution stage. When we peek beneath the covers at the implementation details of the order stage, we see that every order node must receive and maintain a copy of every request that is ordered. We observe that the order stage can add requests to batches and order batches without processing the full requests—a cryptographic hash is sufficient to identify uniquely any ordered request. We consequently modify the authentication stage to cache authenticated requests and send only authenticated request hashes to the authentication stage for ordering. The execution stage then fetches the bodies for all ordered requests prior to executing a batch. The changes to basic operation are depicted in Figure 5.2.

We say a request is *fetchable* if the request body is stored at the authentication stage. Accommodating this change to the protocol requires us to introduce an additional safety and liveness property at the authentication stage.

- **AS2** Every authenticated request is *fetchable*.
- **AL2** If the authentication stage receives a FETCH message from the execution stage for a authenticated request n_c issued by client c, then the authentication stage responds with the request body.

This change also requires us to modify OS1 to ensure that only requests that are both authenticated *and* authenticated are ordered[4].

- **OS1** Only *fetchable* client requests authenticated by the authentication stage are placed into batches, and request n_c issued by client c is placed in at most one batch.

[4]This modification is subtle and its necessity is not immediately apparent. We will revisit this point in section 6.4.

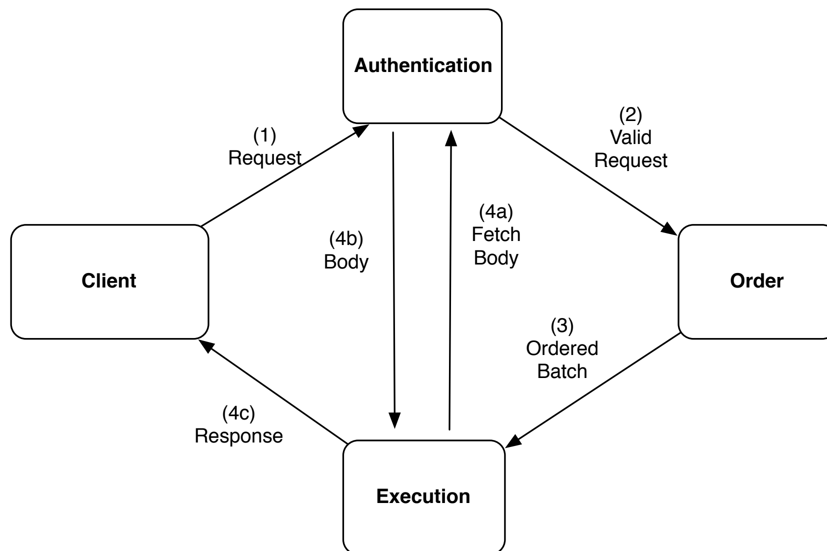

Figure 5.2: Messages exchanged between stages. (1) Clients send requests to the authentication stage. (2) The authentication stage sends validated request hashes to the order stage. (3) The order stage sends ordered batches to the execution stage. (4a, 4b) The execution stage fetches request bodies from the authentication stage. (4c) The execution stage sends responses to the clients. Note that the messages travel through the system in a clockwise fashion.

Additionally, we expand the liveness property EL1 into two components that target fetching request bodies and executing batches separately.

EL1a If the execution stage receives ordered batch n_o *and the last batch it has delivered to the application is* $n'_o < n_o$*, then it fetches the request bodies for requests in batch* n_o *from the authentication stage.*

EL1b *If the execution stage has all of the request bodies for batch* n_o and the last batch it delivered to the application is $n_o - 1$, then the execution stage delivers batch n_o to the application.

Note that EL1 can be acquired by combining the non-italicized portions of EL1a and EL1b.

5.3 Garbage collection and transient crashes

Of the five challenges identified at the beginning of this chapter, two remain unaddressed: (1) stages have finite memory and (2) stages can exhibit transient crashes. The mechanisms used to address these challenges are closely intertwined.

Individual machines have finite memory. The retransmission mechanisms used to mask asynchronous network behaviors can require the authentication and order stages to cache for retransmission an arbitrary number of requests and ordered batches respectively. We address this problem through the combination of (a) checkpoint generation and garbage collection and (b) stage interlocking. Checkpoint generation and garbage collection allow us to periodically eliminate a prefix of the state at each stage, and stage interlocking allows us to prevent one stage from getting too far ahead of, or behind, the others.

Figure 5.3 depicts the state stored by each stage and how the pieces interact; the rest of this section is devoted to a detailed description of the basic approach highlighted here. The order stage takes an *order stage checkpoint* every CP_{interval} batches. We ensure that the order stage always maintains at least one checkpoint of its state and a log of between CP_{interval} and $2 \times CP_{\text{interval}}$ batches ordered since that checkpoint was generated. We ensure that the execution stage maintains an *execution stage checkpoint* that corresponds to each checkpoint stored at the order stage and that the authentication stage has the bodies of all requests ordered in subsequent batches. We coordinate garbage collection at the three stages to ensure

that the authentication stage only garbage collects request bodies when they are no longer needed by the execution stage, and the execution stage garbage collects checkpoints only when they are no longer referenced by the order stage. At a high level, these steps ensure that following a transient crash of one or more stages we can resume operation as if nothing went wrong.

We differentiate between transient and persistent memory. The content of transient memory may be lost during a transient crash; the content of persistent memory persists through a transient crash. We ensure that stages survive transient crashes by recording the checkpoints and associated state in persistent memory. The order stage stores order checkpoints and the log of ordered batches in persistent memory, the execution stage stores execution checkpoints in persistent memory, and the authentication stage stores authenticated request bodies in persistent memory.

In the rest of this section we present a stage-by-stage description of the state required by each stage, the construction of checkpoints, and the timing of accesses to persistent memory. The details are tedious, but the specific design choices directly impact the set of properties that each stage must maintain and consequently have significant impact on the stage replication discussed in Chapter 6. Readers not interested in the discussion may wish to read only the "Additional properties" sections in the following text.

5.3.1 Order stage

State. The basic operation described so far requires the order stage to maintain (1) the log of ordered batches, (2) a table containing information (request and batch identifiers) on the last request ordered for each client, and (3) the next batch identifier to be consumed. The log of ordered batches is required to support batch retransmission required by a lossy network; the last ordered table is required to support appropriate handling of retransmitted client requests; the next batch identifier is used to ensure that there are no gaps or repeats in the sequence of batch identifiers.

The order stage maintains two additional pieces of information: a concise description of the history of ordered batches and the current time. The batch history is a required component of the design of our replicated order stage, discussed in Section 6.2, and it is included in the current discussion for completeness only.

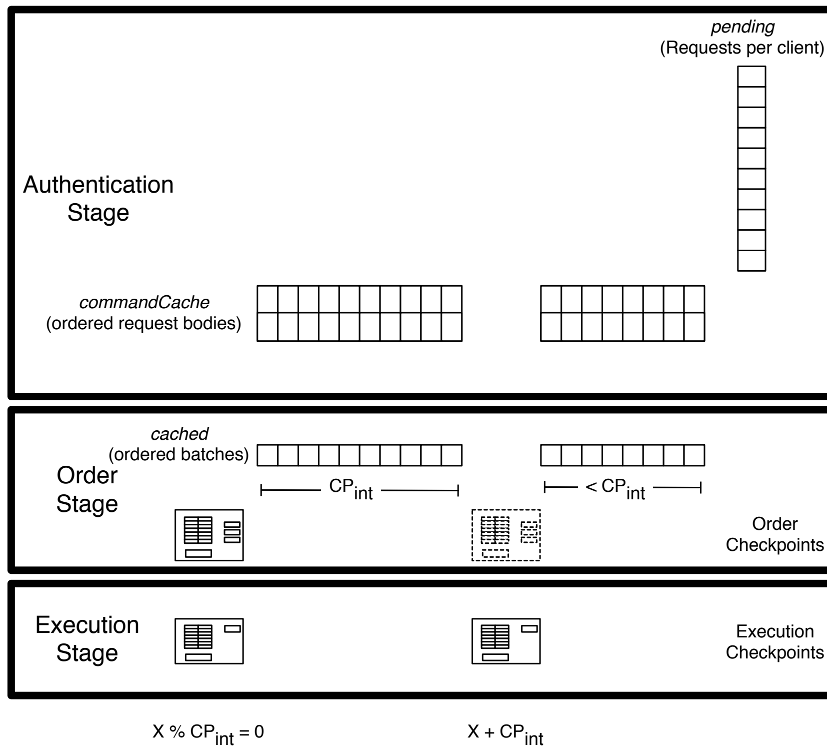

Figure 5.3: Interactions between persistent state at each stage. The state maintained by the other stages depends on the state maintained at the order stage. The order stage maintains one or two checkpoints and between CP_{interval} and $2 \times CP_{\text{interval}} - 1$ ordered batches. The authentication stage maintains every request referenced by an ordered batch stored at the order stage and at most one pending request per client. The execution stage maintains two checkpoints that correspond to order stage checkpoints. Additional details on the contents of the order and execution checkpoints can be found in Figure 5.4 and Figure 5.5 respectively.

The order stage maintains the official system time for the UpRight library and any application replicated with the library. The time is included as part of each ordered batch, and the order stage guarantees that time is monotonically increasing with each batch. Including this time field in each ordered batch is an important part of (a) tolerating transient crashes at the execution stage and (b) facilitating application replication. We discuss the time field in more detail when discussing the execution stage in Section 5.3.2.

We define an order stage checkpoint n_o to be a snapshot of the order stage state taken when all batches $n'_o < n_o$ have been ordered and no batch $n''_o \geq n_o$ has been ordered. An order stage checkpoint is depicted in Figure 5.4: it contains the next batch identifier to be consumed, the last ordered table, the history and time fields, and an execution checkpoint token taken at the same relative point in logical time (i.e. after processing batch $n_o - 1$ and before processing batch n_o). The execution checkpoint token is a concise representation (i.e., a hash) of an execution stage checkpoint. When an order stage checkpoint is initially generated, the execution checkpoint token is initially *null*; that field of the order checkpoint is filled in only after the checkpoint is relayed from the execution stage. We say that an order stage checkpoint is *complete* if it contains the execution checkpoint token and *incomplete* otherwise.

Garbage collection and transient crash recovery. The order stage always maintains a *base* checkpoint taken at n_{CP}, where $n_{\text{CP}} \mod CP_{\text{interval}} = 0$, a *secondary* checkpoint taken at $n_{\text{CP}} + CP_{\text{interval}}$, and a log of between CP_{interval} and $2 \times CP_{\text{interval}}$ batches ordered since the base checkpoint. The base checkpoint and log of ordered batches are *stable*, i.e. stored in persistent memory. The secondary checkpoint may or may not be complete or stable. When the secondary checkpoint becomes complete, it is stored in persistent memory and made stable; only complete checkpoints are stable.

To bound state, the order stage does not order batch $n_{\text{CP}} + 2 \times CP_{\text{interval}}$ unless the secondary checkpoint at $n_{\text{CP}} + CP_{\text{interval}}$ is both stable and complete. This restriction on ordering batch $n_{\text{CP}} + 2 \times CP_{\text{interval}}$ ensures that the order stage is responsible for caching at most $2 \times CP_{\text{interval}}$ batches, each containing at most one request per client. When batch $n_{\text{CP}} + 2 \times CP_{\text{interval}} - 1$ is ordered, three steps are taken:

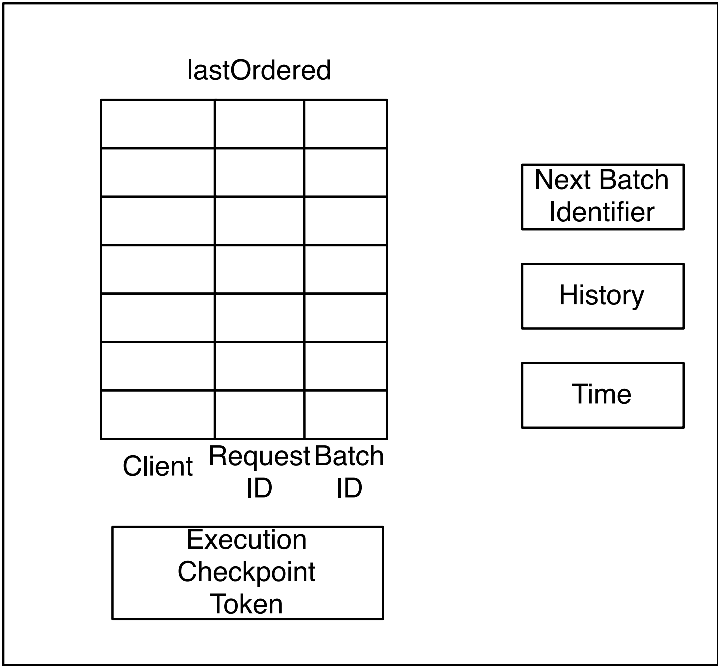

Figure 5.4: Order stage checkpoint.

1. A new secondary checkpoint at $n_{\text{CP}} + 2 \times CP_{\text{interval}}$ is generated.

2. The old secondary checkpoint at $n_{\text{CP}} + CP_{\text{interval}}$ becomes the new base checkpoint.

3. The old base checkpoint at n_{CP} and all batches with identifier $n_o < n_{\text{CP}} + CP_{\text{interval}}$ are garbage collected.

In our implementation, we facilitate garbage collection of persistent memory by storing each stable checkpoint in its own file (checkpoint n_{CP} is stored in the file "order_CP.i" where $i = \frac{n_{\text{CP}}}{CP_{\text{interval}}} \mod 2$) and each block of CP_{interval} ordered batches in a single file (batches n_o through $n_o + CP_{\text{interval}} - 1$ are appended to the file "order_log.i" where $i = \frac{n_{\text{CP}}}{CP_{\text{interval}}} \mod 2$). Note that each ordered batch is recorded to the appropriate log file before it is sent to the execution stage. When checkpoint n_{CP} is garbage collected, the files "order_CP.i" and "order_log.i" are cleared.

The order stage recovers from a transient crash by reading the base checkpoint, secondary checkpoint, and log of ordered batches from persistent memory. If there is no secondary checkpoint stored in persistent memory, then the secondary checkpoint is derived from the base checkpoint and the first CP_{interval} ordered batches in the ordered batch log.

Additional properties. The techniques described above entail an additional safety property maintained by the order stage:

OS4 The order stage always maintains in persistent memory a stable checkpoint at n_o, where $n_o \mod CP_{\text{interval}} = 0$, and $CP_{\text{interval}} \leq i \leq 2 \times CP_{\text{interval}}$ subsequent ordered batches.

The authentication and execution stages rely on this property to determine when it is safe for them to garbage collect state.

Garbage collecting the log of ordered batches prevents the order stage from resending arbitrarily old batches to the execution stage. We consequently modify OL3 to require the order stage to resend only recent batches and add a new liveness property that requires the order stage to send the execution checkpoint descriptor if the execution stage is further behind[5].

[5]These considerations will become more apparent when we discuss execution stage garbage collection and transient crash recovery.

OL3 If the execution stage requests all batches after n_e and the order stage has ordered batches through $n_o \geq n_e$ and $n_e + 1 \geq n_{\text{CP}}$, then the order stage resends all ordered batches from n_e through n_o.

OL4 If the execution stage requests all batches after n_e and the order stage has ordered batches through $n_o > n_e$ and $n_e + 1 < n_{\text{CP}}$, then the order stage instructs the execution stage to load execution checkpoint n_{CP}.

5.3.2 Execution stage.

State. The execution stage maintains a *replyCache* consisting of the last response sent to each client, the identifier n_e of the next batch to be delivered to the application, and a potentially empty set of request bodies for batches that have not yet been delivered to the application. The application is a component of the execution stage; the application state is consequently also part of the state of the execution stage.

We define an execution stage checkpoint n_e to be a snapshot of the execution stage taken when all requests in batches $n'_e < n_e$ have been executed by the application and no request in any batch $n''_e \geq n_e$ has been executed. An execution stage checkpoint contains the *replyCache*, the batch identifier of the next unexecuted batch, and a snapshot of the application state as shown in Figure 5.5.

Garbage collection and transient crash recovery. The execution stage generates a new execution stage checkpoint before delivering batch $n_o \mod CP_{\text{interval}} = 0$ to the application. After generating the checkpoint, the execution stage stores the checkpoint to persistent memory and sends a token (i.e. a hash) that uniquely describes the checkpoint to the order stage. Execution checkpoint n_o is written to file "exec_CP.n_o."

Following a transient crash, the execution stage does nothing until it receives a message from the order stage. Because the execution stage starts off in a default state with $n_e = 0$, it is unlikely to be able to execute the first batch that it receives and will eventually receive a retransmission request. Following receipt of the retransmission request, it notifies the order stage that it has last executed request 0, at which point the order stage instructs the execution stage to load a specific checkpoint by passing the checkpoint token defined by the execution stage. The ex-

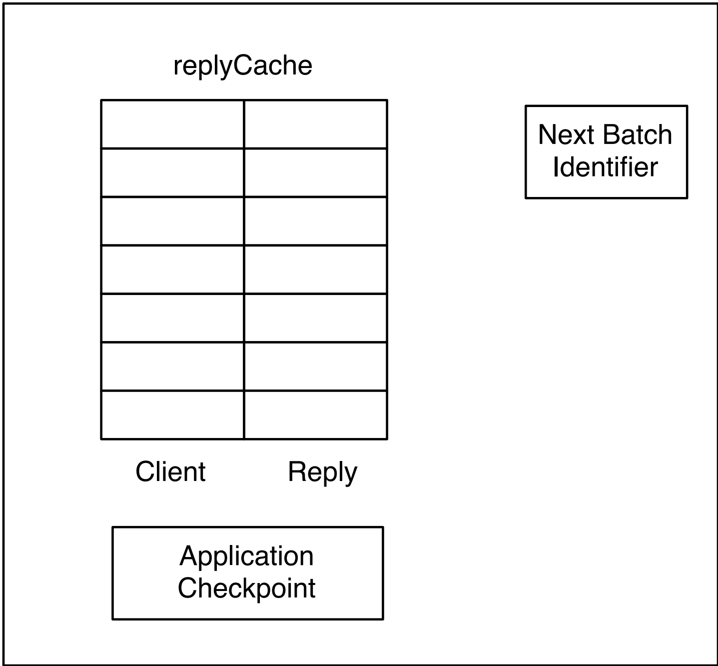

Figure 5.5: Execution stage checkpoint.

ecution stage reads the checkpoint from persistent memory, confirms that the bytes it reads are consistent with the checkpoint token, and then resumes operation using the freshly loaded checkpoint.

The execution stage garbage collects execution checkpoint n_e when it receives an ordered batch with identifier $n_o \geq n_e + 2 \times CP_{\text{interval}}$. This garbage collection is timed to ensure that the execution stage garbage collects a checkpoint only after the order stage has garbage collected any references to that execution checkpoint—ensuring that the order stage will not expect the execution stage to load the checkpoint in the future.

Network efficiency. Note that we do not send the execution checkpoint, but rather a token describing the checkpoint, to the order stage. Sending the token in place of the full checkpoint reduces (a) network traffic and (b) state maintained at the order stage. In Chapter 6 we discuss the implications of other designs, notably storing the full checkpoint or nothing at all in the order stage checkpoint.

Additional properties. We add two new safety properties and a single new liveness property to the execution stage. The safety properties are straightforward. First, the execution stage is required to maintain in persistent memory any checkpoint that it may be instructed to load by the order stage. Second, we require the execution stage to *replay* previously executed batches; the execution of a batch following a checkpoint load must correspond to the execution of that batch that preceded the checkpoint load. The additional liveness property is similarly straightforward: we require the execution stage to load a specified checkpoint on demand.

ES4 The execution stage maintains in persistent memory the execution checkpoint referenced by the order-stage base checkpoint.

ES5 The execution stage provides deterministic and replayable execution of ordered batches.

EL5 If the execution stage receives an instruction to load checkpoint n_e from the order stage, then it loads execution checkpoint n_e.

Application implications. Note that the application property APPS2 is important because the UpRight library relies on log-based rollback recovery [34] to recover

from transient crashes. Without the deterministic execution provided by APPS2 the execution stage could produce different responses when re-executing the set of ordered batches after loading the old checkpoint.

We believe that deterministic execution may be unnecessary if the application can provide fine-grained checkpoints and support checkpoint-based rollback recovery [34]. Fine-grained application checkpoints would allow the execution stage to generate a checkpoint after executing each batch—the system could then agree on the results (responses generated and state reached) of processing each batch before outputting the reponses to the clients. Note that supporting this form of operation would require the library to agree on the result of executing each batch rather than than the order of batches. We leave the exploration of application techniques for efficient fine-grained checkpoints and architectural changes to support speculative execution to future work.

5.3.3 Authentication stage

State. Compared to the order and execution stages, the state maintained by the authentication stage is straightforward, consisting only of the requests that it has authenticated. There are three different types of requests that the authentication stage must store at all times: (1) any request that has been ordered since the current order-stage base checkpoint, (2) any request that has been authenticated and not yet ordered, and (3) the last request authenticated for each client. In most cases the last request authenticated for a client c is either pending or ordered since the base checkpoint; the exception to this rule occurs when a client has been inactive for an extended period of time. The primary challenge in garbage collecting the state of the authentication stage is connected to the maintenance of the second type of requests—requests that have been authenticated but not yet ordered when it is time to perform garbage collection.

The authentication stage maintains three tables in transient memory. The first table, *lastSent*, is indexed by client identifier c and contains the last authenticated request sent to the order stage on behalf of that client. The second table, *pending*, contains up to one tuple $\langle c, n_c, \text{OP} \rangle$ per client and identifies the body of any request authenticated but not yet ordered for client c. When the authentication stage authenticates a client request, it adds the body to the *pending* table and the

authenticated request message sent to the order stage to the *lastSent* table. The third table, *commandCache*, stores one tuple $\langle c, n_c, \text{OP}, n_o \rangle$ per request ordered since the current order-stage base checkpoint. When the authentication stage learns that request n_c issued by client c is ordered in batch n_o, it moves the request body from the *pending* table into the *commandCache*. The *commandCache* is implemented as a set of three distinct tables $commandCache_{\{0,1,2\}}$. Request bodies ordered in batch n_o are stored in $commandCache_i$ where $i = \frac{n_o}{CP_{\text{interval}}} \mod 3$.

Note that the authentication stage effectively maintains 3 checkpoint intervals worth of requests, in contrast to the 2 checkpoint intervals worth of batches maintained by the order stage. Our experience indicates that a slow execution replica is more likely to successfully catch up following a transient crash when the authentication stage caches 3, rather than 2, checkpoint intervals worth of requests. This benefit results from a race condition between the replica successfully fetching the appropriate execution stage checkpoint and the occurrence of the next garbage collection.

Garbage collection and transient crash recovery. When the authentication stage learns that batch n_o has been ordered, it can safely garbage collect any request bodies ordered prior to batch $\frac{n_o - CP_{\text{interval}}}{CP_{\text{interval}}}$. We take a very simple approach to garbage collection. The authentication stage keeps track of the identifier n_o for the maximal ordered batch that it has observed. The first time it learns that batch n'_o has been ordered, where $\frac{n'_o}{CP_{\text{interval}}} > \frac{n_o}{CP_{\text{interval}}}$, it garbage collects $commandCache_i$ where $i = \frac{n'_o}{CP_{\text{interval}}} \mod 3$.

In order to survive transient crashes, authenticated requests must be stored in persistent memory. To that end, the first time a request n_c from client c is authenticated, it is stored to a persistent log of authenticated requests before the authenticated request is sent to the order stage. Garbage collecting the log can be difficult because there may be very little correlation between when a request is authenticated and the ordered batch that it eventually appears in. We consequently maintain a log of authenticated request bodies in a set of three distinct log files organized as a circular buffer: "authentication_log.{0,1,2}." As requests are authenticated and placed in the *pending* and *lasSent* tables, their bodies are recorded into the currently active log file "authentication_log.i" where $i = \frac{n_o}{CP_{\text{interval}}} \mod 3$ and n_o is the maximal batch identifier that the authentication stage has observed. Note that this

log corresponds to the most recently updated $commandCache_i$ and not necessarily the $commandCache$ where the request will eventually be placed. The authentication stage switches from "authentication_log.i" to "authentication_log.j" when it garbage collects $commandCache_j$. At that point, the authentication stage closes "authentication_log.i" and clears the contents of "authentication_log.j." It then dumps the base sequence number of the current checkpoint interval (i.e. $\frac{n_o}{CP_{\text{interval}}} \times CP_{\text{interval}}$) and the contents of the *pending* table to "authentication_log.j"—ensuring that any request placed in $commandCache_i$ is also present in "authentication_log.j" . After logging the *pending* table, the authentication stage resumes processing client requests, recording request bodies to the log file as they are added to the pending table.

Following a transient crash, the authentication stage reconstructs the *pending*, $commandCache$, and $lastSent$ tables from the log files. For each client c, the logged request with maximal n_c is recorded as the entry for $pending[c]$ and a corresponding authenticated request message is placed in $lastSent[n_c]$. If there is no entry for a client, then $lastSent[n_c]$ is left empty. Every request body recorded in file "authentication_log.i" is added to $commandCache_i$, with the exception of requests stored in the *pending* table. Note that requests initially authenticated and recorded in "authentication_log.i" may be placed in a different $commandCache_j$ when they are finally ordered. By recording the *pending* table to the beginning of each log file, we ensure that any request that is ordered during the interval covered by $commandCache_j$ is present in "authentication_log.j." Given a base sequence number *base* recorded at the beginning of "authentication_log.j," the authentication stage does not garbage collect $commandCache_j$ until it after it learns that a batch with identifier $n_o \geq base + 3 \times CP_{\text{interval}}$ has been ordered.

Additional properties. Garbage collecting outdated request bodies slightly modifies the set of requests that are fetchable by the execution stage:

AS2 Every authenticated request *referenced by a batch ordered since the base checkpoint at the order stage or not yet ordered* is fetchable.

We also modify the primary liveness property to ensure increasing client request identifiers or the retransmission of any pending requests.

AL1a If the authentication stage receives a request n_c issued by correct client c *and*

there is no pending request $n''_c < n_c$, then request $n'_c \geq n_c$ is authenticated and sent to the order stage.

AL1b If the authentication stage receives a request n_c issued by correct client c and there is a pending request n'_c, then request n'_c is authenticated and sent to the order stage.

We also modify two liveness properties at the execution stage to ensure that the authentication stage eventually learns that requests have been ordered:

EL1a If the execution stage receives ordered batch n_o and the last batch it has delivered to the application is $n'_o < n_o$, then it fetches the relevant request bodies from the authentication stage *and notifies the authentication stage that the contained requests have been ordered.*

EL3 If the execution stage receives a retransmission instruction for request n_c from c in batch n_o and the last batch executed by the execution stage is $n_e > n_o$, then the execution stage resends the response to the most recent request $n'_c \geq n_c$ executed for client c *and notifies the authentication stage that* n'_c *has been ordered no later than batch* n_e*.*

Fetching a request body implicitly notifies the authentication stage that the request has been ordered, we simply make that implicit knowledge explicit. Retransmission requests received by the execution stage can be caused not only because the client failed to receive a response, but also because the authentication stage did not receive the notification that a pending request has been successfully ordered.

Rate limiting. Faulty clients can issue an arbitrary number of requests and inflate the size of the pending request table. With the goals of robust fault tolerance (Chapter 3) in mind, the authentication stage maintains at most one pending request per client and authenticates at most one request per client per request identifier.

AS3 At most one request per identifier n_c per authorized client c is authenticated.

AS4 When request n_c from client c is authenticated, no request $n'_c > n_c$ has been authenticated and there is no pending request $n'_c < n_c$.

In short, we ensure that there is at most one outstanding request per client waiting to be ordered. Note that we don't require the authentication stage to increment client request identifiers by one even though correct clients obey that restriction. This decision is a nod to the implications of a replicated authentication stage and the reality, discussed in Chapter 6, that client requests can be processed by the system without being processed by every authentication replica.

5.3.4 Client

A client intended to survive transient crashes must store its most recent request in persistent memory. If it does not store the request in persistent memory, then it may not be able to resume operation following a transient crash because it may be unable to recreate a request n_c that was authenticated but not ordered—the authentication stage will reject the request according to AS1.

CS3 Client c stores the most recently issued request in persistent memory.

While it is easy to imagine techniques where the client sends a special "I'm starting over" message after a transient crash, the internal details of how the authentication stage is replicated complicates matters. Specifically, it is possible for a client request to reach only a subset of the authentication replicas before the client suffers the transient crash. Because each authentication replica individually authorizes at most one request per client c per request identifier n_c, this state divergence can prevent any subsequent requests issued by client c from being authenticated.

5.4 Full property list

In this section we consolidate the stage properties describe in Sections 5.1,5.2,and 5.3 into one location. These properties, and the pseudo-code descriptions in Appendix **??**, form the basis of the replicated stage implementations discussed in Chapter 6.

5.4.1 Client Properties

A correct client issues one request per request identifier, consumes request identifiers sequentially, and resends each request until it receives a response. The complete set of properties provided by a correct client follow:

CS1 Each request issued by client c is assigned a unique request identifier n_c starting with 1 and increasing with each subsequent request.

CS2 Client c operates in a closed loop: it does not issue request $n_c > 1$ unless it has received a response to request $n_c - 1$.

CS3 Client c stores the most recently issued request in persistent memory.

CL1 Client c resends request n_c until it receives a response.

5.4.2 Authentication stage properties

The authentication stage authenticates requests to the order stage and caches request bodies as long as they may be required by the execution stage. The authentication stage ensures that only requests from authorized clients are authenticated. The complete set of properties provided by the authentication stage follow:

AS1 Only requests issued by authorized clients are authenticated and every authenticated request is one-step transferable.

AS2 Every authenticated request referenced by a batch ordered since the base checkpoint at the order stage or not yet ordered is fetchable.

AS3 At most one request per identifier n_c per authorized client c is authenticated.

AS4 When request n_c from client c is authenticated, no request $n'_c > n_c$ has been authenticated and there is no pending request $n'_c < n_c$.

AL1a If the authentication stage receives a request n_c issued by correct client c and there is no pending request $n''_c < n_c$, then request $n'_c \geq n_c$ is authenticated and sent to the order stage.

AL1b If the authentication stage receives a request n_c issued by correct client c and there is a pending request n'_c, then request n'_c is authenticated and sent to the order stage.

AL2 If the authentication stage receives a FETCH message from the execution stage for a authenticated request n_c issued by client c, then the authentication stage responds with the request body.

5.4.3 Order stage properties

The order stage places authenticated requests into batches and assigns an execution order to batches. Each distinct request is placed in at most one batch; if it receives a request multiple times then the order stage requests a retransmission of the result rather than ordering the request for execution multiple times. The complete set of properties provided by the order stage follows:

- **OS1** Only fetchable client requests authenticated by the authentication stage are placed into batches, and request n_c issued by client c is placed in at most one batch.
- **OS2** Batches contain one or more requests and are assigned monotonically increasing batch identifiers n_o starting with 1 and increasing by 1 with each subsequent batch. For batches n_o and n'_o with associated times t and t', $n_o > n'_o \rightarrow t > t'$.
- **OS3** If request $n_c > 1$ issued by client c is in batch n_o, then request $n_c - 1$ issued by client c is in batch $n'_o < n_o$.
- **OS4** Order stage always has stable checkpoint at n_o, where $n_o \% CP_{\text{interval}} = 0$, and $CP_{\text{interval}} \leq i \leq 2 \times CP_{\text{interval}}$ subsequent ordered batches.
- **OL1** If the order stage receives unordered authenticated request n_c issued by correct client c, then the order stage places the request in batch n_o and eventually sends a NEXT-BATCH message containing n_o to the execution stage.
- **OL2** If the order stage receives an authenticated request n_c from client c that is already in batch n_o, then it instructs the execution stage to retransmit a response to request n'_c from client c in batch n'_o where $n'_c \geq n_c$ and $n'_o \geq n_o$.
- **OL3** If the execution stage requests all batches after n_e and the order stage has ordered batches through $n_o > n_e$ and $n_e + 1 \geq n_{\text{CP}}$, then the order stage resends all ordered batches from n_e through n_o.
- **OL4** If the execution stage requests all batches after n_e and the order stage has ordered batches through $n_o > n_e$ and $n_e + 1 < n_{\text{CP}}$, then the order stage instructs the execution stage to load execution checkpoint n_{CP}.

5.4.4 Execution stage properties

The execution stage processes batches of requests in the order specified by the order stage. For each processed request, it delivers the result to the client that issued the request. The complete set of properties provided by the execution stage follows:

ES1 Batch n_o is only delivered to the application if the last batch delivered to the application is $n_o - 1$.

ES2 Only ordered batches are delivered to the application.

ES3 Only responses generated by the application are cached or sent to clients.

ES4 Execution stage maintains the execution checkpoint referenced by the order -stage base checkpoint in persistent memory.

ES5 The execution stage has deterministic and replayable execution of ordered batches.

EL1a If the execution stage receives ordered batch n_o and the last batch it has delivered to the application is $n'_o < n_o$, then it fetches the request bodies for requests in batch n_o from the authentication stage and notifies the authentication stage that the contained requests have been ordered.

EL1b If the execution stage has all of the request bodies for batch n_o and the last batch it delivered to the application is $n_o - 1$, then the execution stage delivers batch n_o to the application.

EL2 If the execution stage receives a response from the application, then it stores the response for retransmission and sends the response to the responsible client.

EL3 If the execution stage receives a retransmission instruction for request n_c from c in batch n_o and the last batch executed by the execution stage is $n_e > n_o$, then the execution stage resends the response to the most recent request $n'_c \geq n_c$ executed for client c and notifies the authentication stage that n'_c has been ordered no later than batch n_e.

EL4 If the execution stage receives a retransmission instruction for request n_c from client c in batch n_o and the last batch executed by the execution stage is $n_e < n_o$, then the execution stage informs the order stage that it has missed the batches since n_e.

EL5 If the execution stage receives an instruction to load checkpoint n_e from the order stage, then it loads execution checkpoint n_e.

5.5 Supported optimizations

We support three operation paths in addition to the basic protocol operation described in the previous sections: (a) request pre-fetching between the authentication and execution stages, (b) read-only request execution, and (c) spontaneous server generated replies.

Request prefetching. As a performance optimization, the UpRight library supports request pre-fetching between the authentication and execution stages. When request pre-fetching is enabled, the authentication stage sends the request to the execution stage when it sends the authenticated request hash to the order stage. Request pre-fetching reduces the latency to process requests by ordering and distributing requests in parallel rather than sequentially.

Read only replies. As a performance optimization, the UpRight library supports PBFT's read-only optimization [18], in which a client sends read-only, side-effect-free requests directly to the execution stage and the execution stage processes them without ordering them in the global sequence of requests. If the client receives a response, the client can use the reply; otherwise the request is concurrent with an interfering operation, and the client must reissue the request via the normal path to execute the request in the global sequence of requests. To support this optimization, the client and execution stage must identify read only requests.

Small requests. As a performance operations, the UpRight library does not order hashes of "small" requests (i.e. requests of less than 100B) and instead orders the requests themselves. Small requests are placed directly into the verified request

messages sent to the order stage. When the execution stage receives a batch containing a small request, the request can be executed directly without being fetched from the authentication stage.

Spontaneous replies. Replication libraries are designed around an implicit assumption that all client-server communication follows a simple pattern: clients issue requests and servers generate an immediate response for each processed request. In reality, not all interactions follow this pattern. Specifically, in some systems (a) every client request may not elicit a response from the server or (b) the server can *push* a response to a client without being prodded to do so by a specific request. In the first case, it is straightforward to have the server send a null response back to the client to discharge the obligations of the request-response pattern. The latter case is more difficult, as it is difficult to force the client to issue a request for a response it may not be expecting.

The UpRight library provides unreliable channels for push events. We posit that most client-server systems that rely on push events already cope with the "lost message" case (e.g., to handle the case when the TCP connection is lost and some events occur before the connection to the server can be reestablished), so existing application semantics are preserved. In our implementation, the execution stage includes sequence numbers on push events, sends them in FIFO order, and attempts to resend them until they are acknowledged, but can unilaterally garbage collect any pending push events at any time. The client signals the (presumed existing) application lost-message or lost-connection handler.

5.6 Messages and notation

Messages exchanged among the client and the three server stages are shown in Table 5.1. We augment the message structure and fields with the identity of the stage that sends and receives the message. We expand on the use and meaning of each message in subsequent sections. Details on the byte definitions of these messages can be found in Appendix A.2.

There is a significant amount of notation introduced in the message definitions above. We explain that notation in Table 5.2 and the following text.

We use c to indicate a client. Each time client c issues a command OP, it

Message	Sent by	Received by
$\langle$CLIENT-REQ, $\langle$REQ-CORE, c, n_c, OP$\rangle$, $c\rangle_{\vec{\mu}_{c,\mathcal{F}}}$	client	authentication
$\langle$AUTH-REQ, $\langle$REQ-CORE, c, n_c, hash(OP)$\rangle_{\vec{\mu}_{f,\mathcal{O}}}$, $f\rangle_{\vec{\mu}_{f,\mathcal{O}}}$	authentication	order
$\langle$COMMAND, n_o, c, n_c, OP, $f\rangle_{\mu_{f,e}}$	authentication	execution
$\langle$TOCACHE, c, n_c, OP, $f\rangle_{\vec{\mu}_{f,\mathcal{E}}}$	authentication	execution
$\langle$NEXT-BATCH, $v, n_o, \mathcal{H}, \mathcal{B}, t$, bool, $o\rangle_{\vec{\mu}_{o,\mathcal{E}}}$	order	execution
$\langle$REQUEST-CP, $n_o, o\rangle_{\vec{\mu}_{o,\mathcal{E}}}$	order	execution
$\langle$RETRANSMIT, $c, n_o, o\rangle_{\vec{\mu}_{o,\mathcal{E}}}$	order	execution
$\langle$LOAD-CP, $\mathcal{T}_{\text{cp}}, n_o, o\rangle_{\mu_{o,e}}$	order	execution
$\langle$BATCH-COMPLETE, $v, n_o, \mathcal{C}, e\rangle_{\vec{\mu}_{e,\mathcal{F}}}$	execution	authentication
$\langle$FETCH, n_o, c, n_c, hash(OP), $e\rangle_{\vec{\mu}_{e,\mathcal{F}}}$	execution	authentication
$\langle$CP-UP, $n_o, \mathcal{C}, e\rangle_{\vec{\mu}_{e,\mathcal{F}}}$ $\langle$LAST-EXEC, $n_e, e\rangle_{\vec{\mu}_{e,\mathcal{O}}}$	execution execution	authentication order
$\langle$CP-TOKEN, $n_o, \mathcal{T}_{\text{cp}}, e\rangle_{\vec{\mu}_{e,\mathcal{O}}}$	execution	order
$\langle$CP-LOADED, $n_o, e\rangle_{\vec{\mu}_{e,\mathcal{O}}}$	execution	order
$\langle$REPLY, $n_c, \mathcal{R}, \mathcal{H}, e, \rangle_{\mu_{e,c}}$	execution	client

Table 5.1: Message specification for messages exchanged between stages. The sender and recipients of the messages are indicated.

Notation	Meaning
c	Client identifier
OP	Client command
n_c	Client request identifier
$\mathcal{R}$	Result of processing client command
f	Authentication replica identifier
$\mathcal{F}$	Authentication stage
o	Order replica identifier
$\mathcal{O}$	Order stage
p	"Primary" order replica
e	Execution replica identifier
$\mathcal{E}$	Execution stage
$\mathcal{B}$	Batch of client requests
$\mathcal{C}$	List of client request identifiers
n_o	Batch sequence number
$\mathcal{H}$	History of ordered batches
n_e	Sequence number of last executed batch
$\mathcal{T}_{\text{cp}}$	Execution stage checkpoint
$\mu_{i,j}$	MAC from replica i to replica j
$\vec{\mu}_{o,\mathcal{E}}$	MAC authenticator from replica o to stage $\mathcal{E}$
$\vec{\mu}_{\mathcal{F},\mathcal{O}}$	Matrix signature from stage $\mathcal{F}$ to stage $\mathcal{O}$

Table 5.2: Summary of symbols used and their meanings.

binds the command to a unique identifier n_c.

We differentiate between *stages* and *replicas*. A *stage* refers to the collection of replicas that work together to provide the authentication, order, and execution abstractions. We use f to refer to individual authentication replicas and $\mathcal{F}$ to refer to the authentication stage; o refers to a single order replica and $\mathcal{O}$ refers to the order stage, additionally p refers to a designate order replica also called the primary; e refers to a single execution replica and $\mathcal{E}$ refers to the execution stage.

The order stage collates multiple client requests into a *batch* $\mathcal{B}$. A batch consists of sets of tuples $\langle c, n_c, \textsc{op}\rangle$ and is associated with a non-determinism unit t consisting of the system time and a pseudo random seed[6]. Batches are assigned a unique sequence number n_o by the order stage. The history $\mathcal{H}$ of batches, records the sequence of ordered batches including the time and PRNG seed associated with each batch. The history at batch n_o is computed as $\mathcal{H}_n = \text{hash}(\mathcal{H}_{n-1}, \mathcal{B}_n, t_n)$. We use the history tie successive batches together.

The execution stage reports a sequence number as n_e and periodically reports checkpoints to the order stage through a checkpoint token $\mathcal{T}_{\text{cp}}$. The set $\mathcal{C}$ is composed of tuples $\langle c, n_c\rangle$ corresponding to the last request identifier n_c executed by the execution stage for each client c.

We generically use $\text{hash}(\mathcal{B})$ to indicate a hash of $\mathcal{B}$ and $\mu_{i,j}$ to indicate a MAC authenticated by replica i for verification by replica j. We use $\vec{\mu}_{o,\mathcal{E}}$ to indicate a MAC authenticator generated by replica o for verification by every execution replica and $\vec{\mu}_{\mathcal{F},\mathcal{O}}$ to indicate a matrix signature [3] generated by the replicas in $\mathcal{F}$ for authentication by every replica in $\mathcal{O}$.

5.7 Stage level pseudo-code

In this section we provide pseudo-code for correct clients and the authentication, order, and execution stages. This pseudo-code describes the basic operation for each stage and provides the foundation that the replicated stages discussed in Chapter 6 will emulate. The pseudo-code presented in this chapter implements the stage and client properties listed in Section 5.4 and is presented here for completeness and concreteness. Most readers will want to skip this section.

[6] Note that we explicitly separate the time and pseudo-random seed from the batch as part of the implementation. The time and pseudo-random seed are logically a part of the batch of requests.

```
1   n_c := 0                          \\ next request ID
2   out := ∅                          \\ outstanding client request

4   issueCommand(OP):
5       if out ≠ ∅ then
6           block
7       out := ⟨CLIENT-REQ, ⟨REQ-CORE, c, n_c, OP⟩, c⟩_{μ⃗_{c,F}}
8       store out to persistent memory
9       send out to F
10      start timer

12  on rcv m = ⟨REPLY, n_c, R, H, e, ⟩_{μ_{e,c}}:
13      if m.n_c = n_c then
14          n_c := n_c + 1
15          deliver R to user
16          clear timer

18  on timeout:
19      if out ≠ ∅ then
20          send out to F

22  on recovery:
23      load  out := ⟨CLIENT-REQ, ⟨REQ-CORE, c, n_c, OP⟩, c⟩_{μ⃗_{c,F}} from persistent memory
24      n_c := out.n_c
25      send out to F
26      start timer
```

Figure 5.6: Pseudo-Code for the client

5.7.1 Client operation

Client operation is straightforward and simple. The client issues commands, but before sending the command to the authentication stage it stores the client request message containing the command and the current client request identifier n_c to persistent memory. It continues resending the client request message until it receives a response from the authentication stage. Pseudo-code for the client is shown in Figure 5.6.

In our implementation, clients use an adaptive retransmission policy. When the system starts, the timeout is initially set to 500 ms. Each time a retransmission is required, the timeout is doubled up to a maximum of 4000 ms. When a response is received, the base timeout is set to the maximum of 500ms and the observed latency for the previous request-response pair.

5.7.2 Authentication operation

The authentication stage is responsible for authenticating client requests and caching the request bodies so that they can be fetched by the execution stage. Pseudo-code for the authentication stage is shown in Figure 5.7. We describe the operation of the authentication stage by detailing the state maintained and the processing of each of

the three messages it receives from other participants in the system.

Data structures. The authentication stage maintains three data structures: (1) a set *lastSent* indexed by client identifiers that stores the last validated request message $\langle \text{AUTH-REQ}, \langle \text{REQ-CORE}, c, n_c, \text{hash}(\text{OP})\rangle_{\vec{\mu}_{f,\mathcal{O}}}, f\rangle_{\vec{\mu}_{f,\mathcal{O}}}$ sent to the authentication stage for each client, (2) a set *pending* indexed by client identifiers that stores any validated requests that have been validated but not yet ordered for each client, and (3) a set *commandCache* indexed by a client identifier/client request identifier tuple that stores bodies of validated requests and the batch identifier for the batch containing that request. The *lastSent* set aids in processing retransmitted requests. The *pending* and *commandCache* sets are fundamental in ensuring that any authenticated request body is fetchable.

In addition to the three sets described above, the authentication stage maintains a batch identifier n_f that represents the next batch identifier that the authentication expects to see ordered.

Processing $\langle \text{CLIENT-REQ}, \langle \text{REQ-CORE}, c, n_c, \text{OP}\rangle, c\rangle_{\vec{\mu}_{c,\mathcal{F}}}$**.** The primary task of the authentication stage is authenticating client requests. When the authentication stage receives a request from client c, it first checks if a request from c with request identifier $n'_c \geq n_c$ has already been authenticated by examining the contents of *lastSent*$[c]$. If request $n'_c \geq n_c$ has been authenticated, then the authentication stage resends the authenticated request message stored in *lastSent*$[c]$ to the order stage.

If no request $n'_c \geq n_c$ has been authenticated for client c then the authentication stage confirms that the last request authenticated for c has been successfully ordered by checking *pending*$[c]$. If *pending*$[c]$ is not empty, the the authentication stage discards the client request message, otherwise it authenticates the request and generates a new authenticated request message $\langle \text{AUTH-REQ}, \langle \text{REQ-CORE}, c, n_c, \text{hash}(\text{OP})\rangle_{\vec{\mu}_{f,\mathcal{O}}}, f\rangle_{\vec{\mu}_{f,\mathcal{O}}}$. This message is stored to *lastSent*$[c]$ while the tuple $\langle c, n_c, \text{OP}\rangle$ is stored to persistent memory and *pending*$[c]$.

The authentication stage processes new requests for c as they are received but limits retransmissions to occur at most once per 4000ms per client request.

```
1   lastSent[c] := ∅ \\ last request validated for each
2   client

4   pending[c] := ∅ \\ request validated, but not known
5                               to be ordered, per client

7   commandCache_{0,1,2} := ∅ \\ set of ordeed
8                               commands indexed by client identifier and
9                               client request identifier

11  base_{0,1,2} := 0 \\ initial batch identifier for commandCache_i

13  n_f := 0 \\ expected next batch identifier
14  i := n_f / CP_interval mod 3 \\ log file index

16  on rcv m = ⟨CLIENT-REQ, ⟨REQ-CORE, c, n_c, OP⟩, c⟩_{μ_{c,F}} :
17      if m.n_c ≤ lastSent[c].n_c then
18          send lastSent[c] to O
19      else if pending[c] = ∅ then
20          lastSent[c] := ⟨AUTH-REQ, ⟨REQ-CORE, c, n_c, hash(OP)⟩_{μ_{f,O}}, f⟩_{μ_{f,O}}
21          pending[c] := ⟨c, n_c, OP⟩
22          append pending[c] to authentication_log.i
23          send lastSent[c] to O

25  on rcv m = ⟨BATCH-COMPLETE, v, n_o, C, e⟩_{μ_{e,F}} :
26      if m.n_o ≥ n_f then
27          if (n_f − 1)/CP_interval < m.n_o/CP_interval ∧ n_o > base_i + 3 × CP_interval then
28              i := m.n_o/CP_interval mod 3
29              garbage collect commandCache_i
30              clear authentication_log.i
31              append m.n_o/CP_interval × CP_interval to authentication_log.i
32              ∀c do
33                  if pending[c] ≠ ∅ then
34                      append pending[c] to the authentication_log.i
35          n_f := m.n_o + 1
36      ∀b = ⟨c, n_c⟩ ∈ C do
37          if b.n_c ≥ lastSent[b.c].n_c then
38              if pending[b.c].n_c = b.n_c then
39                  commandCache_i.add(pending[b.c], n_o)
40              pending[b.c] := ∅

42  on rcv m = ⟨FETCH, n_o, c, n_c, hash(OP), e⟩_{μ_{e,F}} :
43      k := m.n_o/CP_interval mod 3
44      OP := commandCache_k.get(c, n_c).OP
45      if hash(OP) = hash(OP) then
46          send ⟨COMMAND, n_o, c, n_c, OP, f⟩_{μ_{f,e}} to e


49  on recover:
50      ∀j ∈ {0, 1, 2} do
51          base_j := initial sequence number from authentication_log.j
52          ∀m = ⟨c, n_c, OP⟩ ∈ authentication_log.j do
53              if pending[m.c] = ∅ ∨ pending[m.c].n_c < m.n_c then
54                  commandCache_j.add(pending[m.c])
55                  pending[m.c] = m
56                  lastSent[m.c] := ⟨AUTH-REQ, ⟨REQ-CORE, m.c, m.n_c, hash(m.OP)⟩_{μ_{f,O}}, f⟩_{μ_{f,O}}
57              else
58                  commandCache_j.add(m)
```

Figure 5.7: Pseudo-Code for the authentication stage to follow.

Processing $\langle \text{BATCH-COMPLETE}, v, n_o, \mathcal{C}, e\rangle_{\vec{\mu}_{e,\mathcal{F}}}$**.** The batch completed message is used to notify the authentication stage that the requests in the specified batch have been ordered. When the authentication stage receives a batch completed message it checks to see if it is safe to perform any garbage collection, performs any relevant garbage collection, and then performs the semantic processing of the message.

Upon receipt of the batch completed message, the authentication stage compares the batch identifier n_o with the next batch it expects to be ordered n_f. If $n_o < n_f$, then the authentication stage proceeds directly to the semantic processing of the message. If $n_o \geq n_f$, then the authentication stage checks if a checkpoint interval boundary occurred between $n_f - 1$ and n_o. If the authentication stage finds that a checkpoint boundary has occurred between $n_f - 1$ and n_o, i.e. $\frac{n_f - 1}{CP_{\text{interval}}} < \frac{n_o}{CP_{\text{interval}}}$, then the authentication stage garbage collects the *commandCache* and transitions to a new log file. Independent of whether garbage collection is appropriate or not, the authentication stage updates n_f to be $n_o + 1$.

The semantic processing of batch completed messages is straightforward. The batch summary $\mathcal{C}$ contains one client id/request identifier tuple $\langle c, n_c\rangle$ per request in the ordered batch n_o. For each request n_c from client c is ordered in batch n_o, the authentication stage compares the identifier of the pending request for that client with n_c. If the pending request identifier $pending[c].n_c = n_c$ then the authentication stage moves the contents of $pending[c]$ to $commandCache[c, n_o]$ and associates the request with batch identifier n_o. The pending request for c is then cleared as long as $pending[c].n_c \leq n_c$.

Note that the pseudo-code allows for gaps in the sequence of client request identifiers even though correct clients do not introduce gaps in their sequence of request identifiers. The abstract authentication stage should not allow for gaps in the sequence of client request identifiers, yet the pseudo-code and our description of the authentication stage does: why? The answer is simple: when the authentication stage is implemented by multiple replicas, it is impossible to ensure that all replicas receive every request without implementing protocol that requires more communication than is strictly necessary. Providing the mechanisms for handling gaps in the client request identifier sequence does not impact the behavior of the abstract (unreplicated) order stage in this discussion, but including those details now simplifies our detailed discussion of replicating the authentication stage in Chapter 6.

Processing $\langle \text{FETCH}, n_o, c, n_c, \mathbf{hash}(\text{OP}), e\rangle_{\vec{\mu}_{e,\mathcal{F}}}$**.** Processing the fetch body message is straightforward. Upon receipt of a fetch body message, the authentication stage pulls the request body stored in $commandCache[c, n_c$. If the body is consistent with the request hash hash(OP) in the message, then the authentication stage sends the body to the execution stage.

Recovery. Following a transient failure, the authentication stage populates the *pending* and *commandCache* sets from the authentication_log.* files. It populates the *lastSent* set with authenticated request messages corresponding to the contents of the *pending* set.

Additionally, following recovery from a transient crash, the authentication stage explicitly delays garbage collection until it has observed ordered batches that span at least 4 checkpoint intervals.

5.7.3 Order operation

The order stage is responsible for placing authenticated requests into batches and assigning an execution order to the batches. Pseudo-code for the order stage is shown in Figure 5.8. We describe the operation of the order stage by detailing the state maintained and the processing of each of the four messages it receives from other participants in the system.

Data structures. The *lastOrdered* set records the client request identifier n_c of the last request ordered for each client n_c. The $\mathcal{B}$ is an incomplete batch of requests that has not yet been communicated to the execution stage. The *cached* log contains between CP_{interval} and $2 \times CP_{\text{interval}}$ consecutive ordered batches. The log is broken up into two pieces; at any point in time either $cached_0$ or $cached_1$ contains CP_{interval} consecutive ordered batches.

In addition to the three sets described above, the order stage maintains the identifier n_o of the next batch to be ordered, the identifier of the current base checkpoint n_{CP}, the current *time* (defined as the time associated with the previously ordered batch), and a binary index *ind*.

Processing $\langle \text{AUTH-REQ}, \langle \text{REQ-CORE}, c, n_c, \mathbf{hash}(\text{OP})\rangle_{\vec{\mu}_{f,\mathcal{O}}}, f\rangle_{\vec{\mu}_{f,\mathcal{O}}}$**.** The primary task of the order stage is creating batches of one or more authenticated request and as-

```
1   lastOrdered := ∅ \\last request identifier ordered for each client
2   cached{0,1} := ∅ \\log of next batch messages indexed by order batch identifier
3   B := ∅ \\batch of requests
4   n_o := 0        \\batch identifier of the next batch to be ordered
5   n_CP := 0        \\batch identifier of the base checkpoint
6   baseCP := ∅      \\ base checkpoint
7   secondaryCP := ∅ \\ secondary checkpoint
8   time := 0                  \\last batch time
9   ind := 0                   \\current log index

11  on rcv ⟨AUTH-REQ, ⟨REQ-CORE, c, n_c, hash(OP)⟩_{μ⃗f,O}, f⟩_{μ⃗f,O} :
12      if lastOrdered[c] ≥ n_c then
13          send ⟨RETRANSMIT, c, n_o, o⟩_{μ⃗o,E} to E
14          return
15      if lastOrdered[c] + 1 = n_c then
16          lastOrdered[c] := n_c
17          B ∪=⟨c, n_c, hash(OP)⟩
18      if B is full then
19          time = System.time
20          t := ⟨time, random⟩
21          cached_ind[n_o] := ⟨NEXT-BATCH, v, n_o, H, B, t, bool, o⟩_{μ⃗o,E}
22          append cached_ind[n_o] to order_log.ind

24          send cached_ind[n_o] to E
25          n_o := n_o + 1
26          if n_o mod CP_interval = 0 then
27              if ¬secondaryCP.isStable() then
28                  wait for secondaryCP.isStable()
29              n_CP := n_o − CP_interval
30              ind := (ind + 1) mod 2
31              garbage collect cached_ind and baseCP
32              clear order_log.ind and order_CP.ind
33              baseCP := secondaryCP
34              secondaryCP := take order checkpoint

36  on rcv m = ⟨CP-TOKEN, v, n_o, T_cp⟩_e μ⃗e,O :
37      if m.n_o ≠ secondaryCP.n_o then
38          discard message and return
39      if secondaryCP.hasExecCP() then
40          discard message and return
41      secondaryCP.addExecCP(T_cp)
42      clear order_CP.ind
43      write secondaryCP to order_CP.ind
44      secondaryCP.makeStable()

46  on rcv ⟨LAST-EXEC, n_e, e⟩_{μ⃗e,O} :
47      if n_CP ≤ n_e < n_o then
48          ∀i ∈ [n_e, n_o)
49              send cached[i] to e
50      if n_e < n_CP then
51          send ⟨LOAD-CP, T_cp, n_CP, o⟩_{μo,e} to e

53  on recover:
54      load cached_i from order_log.i
55      load minimal checkponit in order_CP.0 or order_CP.1 as base checkpoint
56      repopulate remaining variables by replaying contents of cached_0 and cached_1
```

Figure 5.8: Pseudo-Code for the order stage to follow.

signing those batches an order. Upon receipt of a authenticated request, the order stage first checks if the request has already been placed in a batch; if the request has been placed in a batch, then the order stage instructs the execution stage to retransmit the last response to the issuing client c and returns. If the request has not yet been ordered and it corresponds to the next request in sequence for client c, then it is added to the pending batch of requests $\mathcal{B}$ and the *lastOrdered* record for c is updated to n_c. If the batch $\mathcal{B}$ is sufficiently full, then the order stage sets the tuple t to contain the system time to be used when executing the batch of requests and a random PRNG seed. The order stage ensures that time increases with successive ordered batches, i.e. if $n'_o > n_o$ then $t' > t$. The batch $\mathcal{B}$ and tuple t are placed in ordered batch message n_o. The next batch message is added to $cached_{ind}$ and appended to ordered_log.*ind* before it is sent to the execution stage. The next batch identifier to be used is incremented by 1, $n_o := n_o + 1$ and the order stage waits for the next request to arrive.

After incrementing the next batch identifier, the order stage checks to see if it has reached a checkpoint interval. If $n_o \mod CP_{\text{interval}} = 0$, then the order stage has arrived at a checkpoint interval and it is time for garbage collection. The order stage waits for the checkpoint at $n_o - CP_{\text{interval}}$ to become stable, at which point it (1) increments *ind*, (2) designates checkpoint $n_o - CP_{\text{interval}}$ as the new base checkpoint, (3) garbage collects $cached_{ind}$ and the old base checkpoint, (4) clears files associated with garbage collected state, and (5) generates a secondary checkpoint at n_o.

Processing $\langle\text{CP-TOKEN}, v, n_o, \mathcal{T}_{\mathbf{cp}}\rangle_{e}\vec{\mu}_{e,\mathcal{O}}$. The order stage receives CP messages from the execution stage that contain a token $\mathcal{T}_{\text{cp}}$ describing the execution checkpoint at n_o. Upon receipt of the execution checkpoint token, the order stage adds $\mathcal{T}_{\text{cp}}$ to the order checkpoint at n_o and stores the order checkpoint to order_CP.*ind*.

Processing $\langle\text{LAST-EXEC}, n_e, e\rangle_{\vec{\mu}_{e,\mathcal{O}}}$. The execution stage sends last executed messages when it detects that the network is not behaving reliably due to dropping/delaying/reordering a subset of next batch messages. When the order stage receives a last executed message, it responds with the ordered batches with identifiers in the range n_e to n_o exclusive.

If the last batch executed by the execution stage n_e is smaller than the base checkpoint maintained by the order stage $base_{CP}$, then the order stage instructs

the execution stage to load the execution checkpoint described by $\mathcal{T}_{\text{cp}}$ stored in order checkpoint $base_{CP}$. This scenario seems far fetched, but can occur when the execution stage suffers a transient crash. The scenario can also occur during normal operation when the execution stage is replicated because the network (and faulty order replicas) cannot be relied on to deliver messages to all order replicas in a timely fashion.

Recovery. Following recovery from a transient crash, the order stage sets the base checkpoint to be the earliest checkpoint contained in order_CP.0 or order_CP.1. The order stage then reads the contents of order_log.i into $cached_i$ for $i \in \{0, 1\}$ and updates *lastOrdered* to be consistent with the logged next batches as described in Section 5.3.1.

5.7.4 Execution operation

The execution stage is responsible for delivering batched requests to the application in the linearized order specified by the order stage and relaying the response to each request to the client that issued the request. Pseudo-code for the execution stage is shown in Figure 5.9. We describe the operation of the execution stage by detailing the state maintained and the processing of each of the four messages it receives from other participants in the system.

Data structures. The execution stage maintains a *replyCache* of the most recent response sent to each client. It also maintains a collection of sets of *batchCommands*. Each set in *batchCommands* corresponds to the set of request bodies specified for an ordered batch and is augmented by the designated system time and PRNG seed.

The execution stage additionally maintains an identifier n_e of the next batch to be executed.

Processing $\langle \text{NEXT-BATCH}, v, n_o, \mathcal{H}, \mathcal{B}, t, \mathbf{bool}, o \rangle_{\vec{\mu}_{o,\mathcal{E}}}$. When the execution stage receives an ordered batch, it first compares the batch identifier n_o with the identifier n_e of the next unexecuted batch in the sequence. If $n_o < n_e$ then the execution stage discard the next batch message. If $n_o \geq n_e$, then it notifies the authentication stage that the batch is complete and fetches the bodies of all requests contained in the batch.

```
1   n_e := 0  \\ identifier of the next batch to execute
2   replyCache := ∅ \\ last reply sent to each client
3   batchCommands := ∅ \\ per batch set of request bodies, keyed by batch identifier
4   state := ∅ \\ application state

6   on rcv  m = ⟨NEXT-BATCH, v, n_o, H, B, t, bool, o⟩_{μ_{o,E}} :
7       if m.n_o < n_e then
8           discard and return
9       else if m.n_o ≥ n_e then
10          batchCommands[m.n_o].setTimeAndPRNG(t)
11          batchCommands[m.n_o].setCommands(B)
12          C := ∅
13          ∀b = ⟨c, n_c, hash(OP)⟩ ∈ B do
14              batchCommands ∪=⟨b.c, b.n_c⟩
15          send ⟨BATCH-COMPLETE, m.v, m.n_o, C, m.e⟩_{μ_{e,F}} to F
16          ∀⟨c, n_c hash(OP)⟩ ∈ B
17              send ⟨FETCH, n_o, c, n_c, hash(OP), e⟩_{μ_{e,F}} to F

19  on rcv ⟨COMMAND, n_o, c, n_c, OP, f⟩_{μ_{f,e}} :
20      batchCommands[n_o].add(c, n_c, OP)
21      while batchCommands[n_e].isComplete do
22          ⟨state, responses⟩ := app.exec(state, C[n_e])
23          batchCommands[n_e] := ∅
24          ∀r = ⟨c, n_c, R⟩ ∈ responses
25              replyCache[r.c] := ⟨REPLY, r.n_c, r.R, H, e, ⟩_{μ_{e,r.c}}
26              send replyCache[r.c] to r.c
27          n_e := n_e + 1
28          if n_e mod CP_interval = 0 then
29              CP_app := app.takeCP(state)
30              CP_exec := take execution checkpoint n_e
31              CP_exec.setAppCP(CP_app)
32              record CP_exec to exec_CP.n_e
33              T_cp := hash(CP_exec)
34              ∀i ≤ n_e − 2 × CP_interval : ∃ exec_CP.{n_e} do
35                  delete exec_CP.n_e
36              send ⟨CP-TOKEN, n_e, T_cp, e⟩_{μ_{e,O}} to O

38  on rcv ⟨RETRANSMIT, c, n_o, o⟩_{μ_{o,E}} :
39      if n_o ≥ n_e then
40          send ⟨LAST-EXEC, n_e, e⟩_{μ_{e,O}} to O
41      if n_o + 1 = n_e then
42          send ⟨CP-TOKEN, CP_interval × (n_e / CP_interval), T_cp, e⟩_{μ_{e,O}} to O
43      if n_o < n_e then
44          C := ∅
45          ∀m = ⟨REPLY, n_c, R, H, e, ⟩_{μ_{e,c}} ∈ replyCache do
46              C ∪=⟨m.c, m.n_c⟩
47          send ⟨BATCH-COMPLETE, v, n_e − 1, C, e⟩_{μ_{e,F}} to F
48      send replyCache[c] to c

50  on rcv  ⟨LOAD-CP, T_cp, n_o, o⟩_{μ_{o,e}} :
51      CP_exec := load checkpoint from exec_CP.n_o
52      if hash(CP_exec) = T_cp then
53          load CP_exec
54          state := app.loadCP(CP_exec.getAppCP())
55          set n_e = n_o
```

Figure 5.9: Pseudo-Code for the execution node to follow.

Note that notifying the authentication stage that the batch is complete and fetching request bodies from the authentication stage are actions based on sending two distinct messages. Using two distinct messages is unnecessary at the intra-stage level of the protocol, but becomes an important concern when the authentication and execution stages are replicated—while it is certainly sufficient for every authentication replica to send every request body to every execution replica, it is not necessary. Using the two distinct messages, one to notify the authentication stage that requests have been ordered and the other to explicitly fetch the requests allows us to limit the number of times a request body is sent over the network. We discuss these concerns in more detail in Chapter 6.

Processing $\langle\text{COMMAND}, n_o, c, n_c, \text{OP}, f\rangle_{\mu_{f,e}}$. Upon receipt of a request body message, the execution stage adds the body to the set of bodies it has gathered for batch n_o. Batch n_o is *complete* if the execution stage has bodies for every request in the batch. After adding a body to the batch, the execution stage checks if batch n_e, the first batch it has not yet executed, is complete. If batch n_e is complete, then the execution stage delivers the batch (including time and PRNG) to the application, stores the responses in the *replyCache* and sends the responses to the appropriate clients before incrementing n_e by one. The authentication stage continues this process until batch n_e is not complete, either because the execution stage is missing one or more request bodies or because the execution stage has not yet received the ordered batch n_e from the order stage.

Before executing batch $n_e \% CP_{\text{interval}}$, the execution stage takes a checkpoint of the execution state, records the checkpoint to persistent memory, and sends a token $\mathcal{T}_{\text{cp}}$ describing the checkpoint to the order stage.

Processing $\langle\text{RETRANSMIT}, c, n_o, o\rangle_{\vec{\mu}_{o,\mathcal{E}}}$. The retransmission message contains two important fields, the client c that needs a response and the next batch identifier n_o that will be used by the order stage. Upon receipt of a retransmission message, the execution stage sends client c the last response stored in the *replyCache* for c. Additionally, if n_o is at least the next unexecuted batch identifier n_e then the execution stage notifies the order stage that it has not yet executed any batch with identifier $n'_o \geq n_e$ and waits for the appropriate batches to be retransmitted; if $n_o + 1 = n_e$, then the execution stage sends the execution stage checkpoint to the

order stage; if $n_o < n_e$ then the execution stage sends a special batch completed message to the authentication stage. This message asserts that every request with a response in the $replyCache$ is ordered in batch $n_e - 1$ (or earlier).

Processing $\langle \text{LOAD-CP}, \mathcal{T}_{\mathbf{cp}}, n_o, o \rangle_{\mu_{o,e}}$**.** When the execution stage receives a load checkpoint message, it loads the execution checkpoint described by $\mathcal{T}_{\text{cp}}$. We implement the checkpoint token $\mathcal{T}_{\text{cp}}$ as a hash of the byte representation of the execution checkpoint.

Recovery. The execution stage does not do anything to recover from transient crashes. It simply begins operation as if it is starting from a fresh slate and waits for messages form the order stage.

5.8 Conclusion

This chapter describes the interactions between correct stages in the UpRight library. Because we assume the stages are correct, and not ideal, the interactions between correct stages accounts for the possibility that clients or the network may be faulty, the reality that network and storage resources are finite, and the threat of transient power outages. The interactions between stages define the properties that each stage must fulfill in order to reliably replicate an application. We discuss the challenges of replicating each stage to sustain the requisite properties despite failures in the next chapter.

The replicas of each stage must be deployed on separate machines to receive the benefits of fault tolerant replication. Although the stages are described as logically separate entities, the replicas implementing each stage need not be physically separate. For example, one machine can host an authentication and order replica while another hosts an authentication and execution replica while yet another machine hosts only an execution replica.

Chapter 6

UpRight Replication

Chapter 5 describes the interactions between the stages of the UpRight architecture assuming that each replicated stage provides the abstraction of a single correct machine. In this chapter we focus on the intra-stage protocols required to discharge the assumption that an individual stage is correct. We consider a replicated stage to be correct if it is up—i.e. ensures the liveness properties specified in Chapter 5—despite at most u arbitrary failures; and right—i.e. ensures the safety properties specified in Chapter 5—despite at most r commission failures.

When discussing the replication of each stage, there are three key questions that we must address. How do the replicas within a stage coordinate with each other? How does replicating one stage impact the other stages? How many replicas are required to implement each stage?

The answer to all three questions is closely tied to the challenge of solving asynchronous consensus. The combination of stages described in Chapter 5 can be viewed as a sequence of three consensus protocols. In the first instantiation of consensus, clients propose requests, the authentication accepts and authenticates the requests, and the order stage learns the authenticated requests. In the second instantiation of consensus, the authentication stage proposes authenticated requests, the order stage accepts and orders the requests in batches, and the execution stage learns the ordered request batches. In the third instantiation of consensus, the order stage proposes ordered batches of requests, the execution stage accepts the ordered batches and executes them in order, and the clients learn the results of executing requests in the sequence of ordered batches.

Stage	Replication requirements
Authentication	$u + \max\{u, r\} + r + 1$
Order	$2u + r + 1$
Execution	$u + \max\{u, r\} + 1$

Table 6.1: Summary of stage-level replication requirements.

We consequently base our design and implementation of each stage on consensus protocols. Although consensus is a well known and extensively studied problem, it is important to note that consensus protocols are not created equal. Both the replication requirements and the acceptor-acceptor and proposer-acceptor-learner communication are influenced by (a) the identity and number of proposers and learners, (b) the desired number of communication steps between proposing and learning, and (c) the semantics of the values being learned. While $2u + r + 1$ (i.e., $3f + 1$ when $f = u = r$) replicas are generally sufficient to solve consensus, there are interesting configurations (e.g., a single unfailing proposer) that require fewer acceptors ($u + r + 1$). Similarly, there are semantics (e.g., a proposed value must be one-step transferable and authenticated via MACs) that may require additional acceptors ($u + 2r + 1$) [32, 60, 56, 68].

Table 6.1 shows the replication requirements for the authentication, order, and execution stages. The order stage requires the standard $2u + r + 1$ replicas. The execution stage requires fewer replicas—$u + \max\{u, r\} + 1$ to be precise. This number is impacted by two considerations that will be explored in this chapter: (1) the order stage acts as a single unfailing proposer and (2) execution checkpoints require indirect learning. Indirect learning occurs any time a hash of data, rather than the data itself, is passed from one stage to another. The authentication stage instead requires $u + \max\{u, r\} + r + 1$ replicas. The core replication requirements of the authentication stage are based on the same factors as the execution stage, with the additional requirement that learned values must be one-step transferable.

In the rest of this chapter we expand on the design of each stage and the specifics of the consensus problem that each stage solves. In Section 6.1 we discuss relevant background on asynchronous consensus, paying specific attention to environments that do not require the standard $2u + r + 1$ acceptors. In Section 6.2 we discuss the mapping of the order stage to consensus and thedetails of implementing a replicated order stage with $2u + r + 1$ replicas. We begin the discussion of repli-

cated stages with the order stage because it is most similar to previous work. In Section 6.3 we discuss the mapping of the execution stage to consensus and details of implementing a replicated execution stage with $u + \max\{u, r\} + 1$ replicas. In Section 6.4 we discuss the mapping of the authentication stage to consensus and the details of implementing a replicated authentication stage with $u+\max\{u, r\}+r+1$ replicas. In Section 6.5 we present microbenchmark experiments that explore the performance characteristics of our prototype implementation of the UpRight library. In Section 6.6 we discuss the costs and benefits of maintaining the logical separation when stages are replicated.

6.1 Consensus background

Recall that we briefly introduced Paxos style consensus [53, 54, 56] in Chapter 2 and subsequently used consensus as a concrete foundation for discussing the UpRight failure model. To recap, the participants in a consensus protocol are divided into three categories based on their role in the system. Proposers propose values, acceptors accept proposed values, and learners learn accepted values.

A consensus protocol is correct if its safety properties hold despite up to r commission failures and its liveness properties hold despite up to u total failures. The three consensus safety properties are: (1) only proposed values are accepted, (2) at most one value is accepted, and (3) non-faulty learners only learn accepted values. The single consensus liveness property is: if a non-faulty proposer proposes a value during a synchronous interval then non-faulty learners eventually learn a value.

We highlight consensus at this point in the thesis for two reasons. First, because RSM protocols are traditionally based on consensus protocols, it is important to understand consensus before discussing the details of replication protocols. Second, the number of replicas required to implement consensus varies with (a) the configuration of proposers, acceptors, and learners, (b) the targeted communication steps, and (c) the semantics of the values being learned.

Consensus and state machine replication. RSM protocols including PBFT [18], Paxos [53], Zyzzyva [49], and many others are traditionally built around repeatedly executing a consensus protocol that is used to order requests for processing. During

the ith instance of consensus, the replicas assign the sequence number i to a request. The replicas then execute the requests in the specified order and relay the result of executing the request to the client that issued the request before moving on to the $i+1^{\text{st}}$ instance of consensus. When the client receives a response, it explicitly learns the result of executing its request as the ith request in the sequence and implicitly learns the relevant impact of the previous $i-1$ requests.

Consensus replication requirements. As mentioned in the previous section, the number of acceptors (i.e. replicas) required to implement consensus is not always $2u+r+1$[1]. While $2u+r+1$ replicas are generally sufficient to solve asynchronous consensus, this number can increase or decrease based on the specific semantics and configuration of replicas for that instance of consensus [32, 60, 56, 68]

If there is exactly one proposer and that proposer cannot fail, then $u+r+1$ replicas are sufficient to solve consensus [32, 60, 56]. The intuition for the reduced costs is straightforward. First, a single proposer that cannot fail can be trusted to send the same messages, in the same order and with appropriate message identifiers, to each acceptor. Second, a correct acceptor can be trusted to accept messages in the order they were sent by the proposer. Third, a learner that receives matching values from $r+1$ distinct acceptors knows that at least one acceptor is non-faulty; it can then be confident that no other value will be accepted for that sequence number and proceed to learn the value. Note that this implies that $r+1$ replicas are sufficient to ensure the safety properties. An additional u replicas are required to ensure that a value can be learned, i.e., that a quorum of $r+1$ correct replicas exists, despite up to u total failures.

The semantics of the value being learned can increase the replication requirements. Continuing to consider a single correct proposer, we explore two specific semantics of learned values—*indirect learning*, i.e. a hash of a value is learned rather than the value itself, and *one-step transferable* with MAC authentication, i.e. the initial learner can teach other learners a value authenticated with MACs.

In the context of indirect learning, the learner can be sure that the hash is correct after receiving the value from $r+1$ distinct acceptors but has no assurance that the actual value will be available for use in the future. If it is important that the value itself be fetchable in the future, then the learner must receive the hash from

[1]Recall that $2u+r+1$ is equivalent to $3f+1$ when $u=r=f$.

$u+1$ distinct acceptors. Combining these two concerns, the learner must receive the hash from $\max\{u, r\} + 1$ distinct acceptors to be sure that the hash is correct and the underlying value is fetchable. Ensuring that indirect learning is always possible requires an additional u replicas for a total of $u + \max\{u, r\} + 1$.

One-step transferability requires any correct learner a that learns a value directly from the acceptors to *teach* correct learner b that same value. In order for a to teach a learned value to b, a must provide b with the value and sufficient proof for b to believe that the value was accepted. With a single correct proposer, a can learn a value when it receives the value from $r + 1$ distinct acceptors. If the messages are authenticated using public key cryptography, then a can pass the $r + 1$ signatures and the value to b and know that b, if correct, will also learn the value. If, however, messages are authenticated with MAC authenticators, then a cannot be sure that b will successfully authenticate all $r + 1$ MAC authenticators and recognize that the value has been accepted—some subset of the acceptors may be faulty. If a receives the value from $2r+1$ replicas, on the other hand, it can provide the value and the set of $2r + 1$ MAC authenticators to b and know that b will successfully authenticate at least $r+1$ authenticators and subsequently learn the value. An additional u replicas are required to ensure that a can always teach a learned value to b[2].

Note that matrix signatures [3] provide a general mechanism for implementing one-step transferability. The discussion here clarifies the relationship between consensus and matrix signatures. We initially developed matrix signatures in the traditional context of f Byzantine failures and observed that matrix signatures, like standard consensus, require $3f+1$ replicas. We now see that matrix signatures solve a specialized consensus problem with a replication requirement of $u+2r+1$ replicas that differs from the traditional $2u + r + 1$.

6.2 Replicated order stage

The order stage is responsible for placing requests in batches and selecting a linearized batch order. The order stage also tracks the recent execution-stage checkpoints used as part of (1) keeping state at each stage finite and (2) recovering from

[2]We note that one-step transferability is the core property provided by digital signatures in Chapter 3.3.

stage level transient crashes. We view both of these activities as instances of consensus, where the order stage acts as the acceptors. We refer to the first instance as *normal-operation* and the second as *checkpoint-operation*.

During normal-operation, the authentication stage proposes authenticated requests to the order stage, the order stage accepts the requests by placing them into batches and assigning an order to the batches, and the execution replicas learn the linearized sequence of request batches. Note that because the authentication stage acts as a proxy for clients that actually issue requests and does not provide any cross-client coordination, we view it as acting as multiple proposers—one proposer per client[3]. Thus, the normal-operation consensus problem corresponds to the standard configuration with multiple proposers and multiple learners.

During checkpoint-operation, the execution stage proposes execution checkpoints to the order stage, the order stage accepts and stores the checkpoint, and the execution replicas may (or may not) learn the checkpoint. Note that the execution stage acts as a single proposer while individual execution replicas learn the checkpoint.

The rest of this section details our design for the replicated order stage. Section 6.2.1 describes the Zyzzyvark protocol, a PBFT-like [18] consensus protocol, used for normal-operation. Section 6.2.2 describes our approach to piggybacking checkpoint-operation onto Zyzzyvark's internal checkpointing mechanisms. Section 6.2.3 describes how the inter-stage messages sent to and from the order stage fit into our design and how replicating the order stage impacts the behavior of the authentication and execution stages. Section 6.2.4 describes how the replicated order stage fulfills the properties of a correct order stage described in Chapter 5.

6.2.1 Normal-operation—Zyzzyvark

Normal-operation maps to a standard configuration of consensus with multiple proposers and multiple acceptors that is comparable to the consensus problem solved by the PBFT lineage of RSM protocols [18, 24, 26, 50, 49, 92, 104, 107]. We could, in principle, use a protocol like PBFT [18], Zyzzyva [49], or Aardvark [24] as the basis for the order stage. We instead rely on a new replication protocol called Zyzzyvark. We do not introduce any fundamentally new ideas or insights in the design of

[3]Semantically, that the authentication stage asserts "Client c said X", not "Client c said X before client c' said Y."

Zyzzyvark. We instead combine key ideas from previous protocols to get a simple and robust protocol design.

Zyzzyvark, like its predecessors, is based on three subprotocols: agreement, checkpointing, and view change. The agreement subprotocol is used to batch and order requests. One replica is designated the primary of the current view and is responsible for leading the replicas through a standard three phase commit protocol to agree on the order and contents of request batches. The checkpoint subprotocol is used to coordinate checkpoints across replicas and allow the garbage collection of old batches and checkpoints. The view change protocol is used to replace the current primary and transition to a new view led by a new primary. As part of transitioning to a new view v, the view change protocol must ensure that the starting state for view v reflects all batches ordered in previous views $v' < v$.

In this section we provide an overview of how the Zyzzyvark protocol works and highlight the ways in which Zyzzyvark differs from previous protocols. We refer readers interested in technical proofs and detailed description of agreement and view change protocols to PBFT [18] and Zyzzyva [49].

Replication requirements and quorum size

Previous protocols have been designed to be safe and live despite up to f Byzantine failures. We design Zyzzyvark to be safe despite up to r commission failures and live despite up to u Byzantine failures.

Translating a protocol described in the language of traditional Byzantine fault tolerance to the language of UpRight fault tolerance is relatively straightforward, requiring only the relabeling of quorum sizes in the system. We identify three distinct quorum sizes as *small*, *medium*, and *large* quorums. In most systems, the protocols are described with the explicit assumption that the minimum $3f + 1$ replicas are used. In that context, small quorums have size $f + 1$ and correspond to the smallest quorum guaranteed to contain at least one correct replica; medium quorums have size $2f + 1$ and correspond to the largest quorum that a replica can wait for without endangering liveness; large quorums have size n and contain every replica. Translating these quorums definitions to UpRight is straightforward: a small quorum has size $r + 1$, a medium quorum has size $n - u$, and a large quorum has size n.

Agreement

We begin by describing the PBFT-like agreement protocol that is the core of Zyzzyvark. The protocol begins when the authentication stage, on behalf of an authorized client c, sends an AUTH-REQ message to the designated *primary* order replica. The primary adds the authenticated request contained in the AUTH-REQ message to a batch; if the batch is "full" or sufficient time has passed since the last batch was formed, the primary sends a PRE-PREPARE message containing the finalized batch to the other replicas. Each replica verifies that the batch is well-formed—that is that the batch identifier is the next in sequence, that the time associated with the batch is larger than the time associated with the previous batch, and that all requests in the batch are (a) issued on behalf of an authorized client, (b) the next in sequence for that client, and (c) have not been placed in a previous batch). If the batch is well-formed then the replica sends a PREPARE message to the other replicas. Upon receipt of a medium quorum of matching PREPARE messages each replica sends a COMMIT message to the other replicas. Upon receipt of a medium quorum of matching COMMIT messages each order replica sends a NEXT-BATCH message to the execution stage, notifying the execution stage that the batch has been ordered. The execution stage accepts the batch as ordered when it receives a small quorum of matching notifications. This basic communication pattern is employed by PBFT [17] and is shown in Figure 6.1.

A straightforward optimization of the basic pattern described above is tentative agreement [18]. Under tentative agreement, the replicas send a tentative batch (TENT-BATCH) to the execution stage after receiving the quorum of PREPARE messages as shown in Figure 6.2. The execution stage accepts the batch as ordered upon receipt of a medium quorum of matching TENT-BATCH messages. The primary contribution of the Zyzzyva work is speculative ordering [49]. When speculative ordering is employed, replicas notify the execution stage that a batch is speculatively ordered after receiving the PRE-PREPARE message from the primary as shown in Figure 6.3. The execution stage accepts a batch as ordered when it receives a large quorum of SPEC-BATCH messages. Enabling speculative ordering requires the order replicas to agree not just on the contents of the next batch, but also on the history of batches that have been ordered—replicas only accept a PRE-PREPARE message if (a) the batch is well formed, (b) the batch is specified as the next batch in the

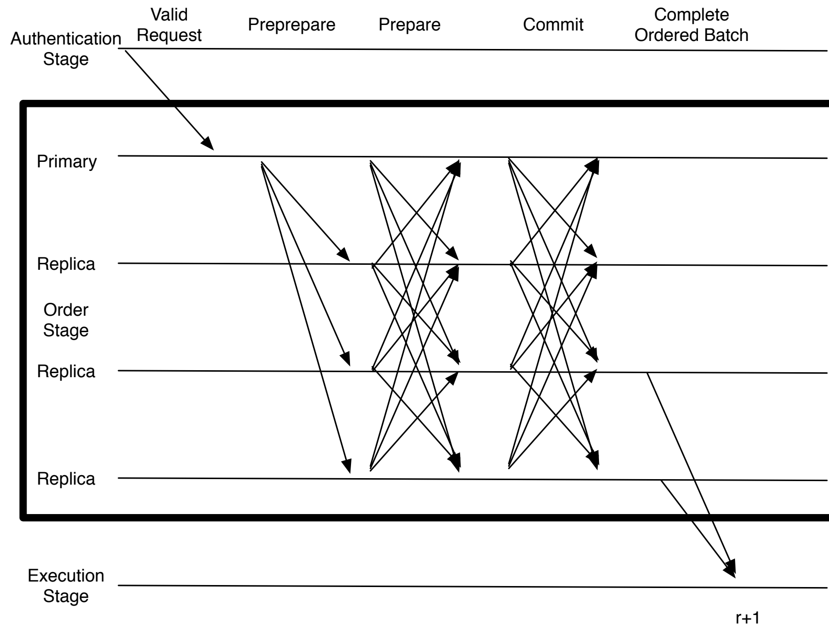

Figure 6.1: Basic communication pattern for complete agreement.

sequence, and (c) the history $\mathcal{H}$ contained in the PRE-PREPARE message summarizes the sequence of batches that the replica has observed.

The Zyzzyvark protocol makes use of both speculative and traditional ordering. By default, the protocol relies on speculative ordering, and replicas do not exchange PREPARE or COMMIT messages. The primary can, however, initiate the traditional three phase order protocol at any time. This may be appropriate and/or necessary if the primary believes that another replica is faulty, or if one replica has requested a view change but the other replicas have not yet joined in the insurrection.

Note that one benefit of including the current batch history with every batch is that *committing* (i.e. gathering a quorum of COMMIT messages) a batch n_o with history $\mathcal{H}$ implies that all batches $n'_o < n_o$ whose histories are prefixes of $\mathcal{H}$ are also committed. This observation extends to the execution stage processing of next-batch messages. Recall that the execution stage waits for n speculative next-batch messages, $n - u$ tentative next-batch messages, or $r + 1$ committed next-batch messages for each batch and that each next batch message contains a batch and the

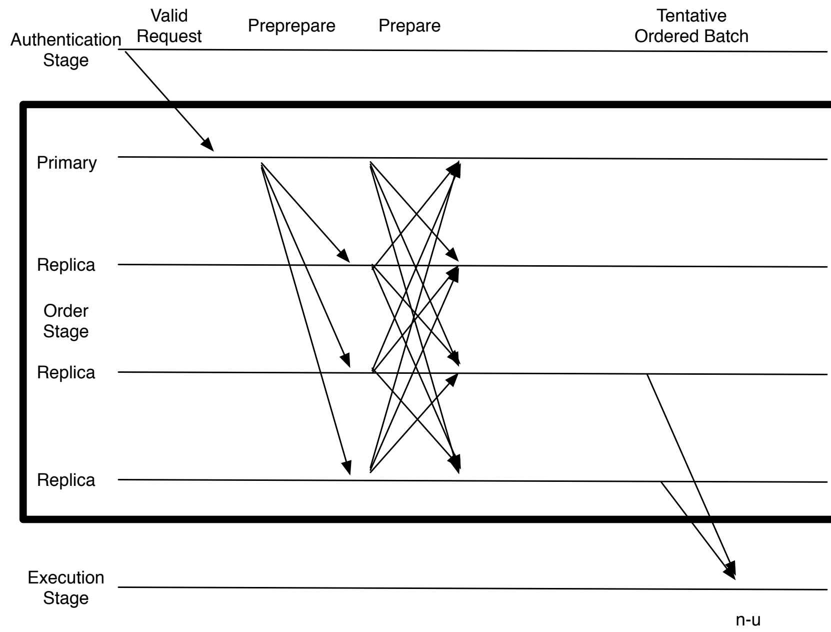

Figure 6.2: Basic communication pattern for tentative agreement.

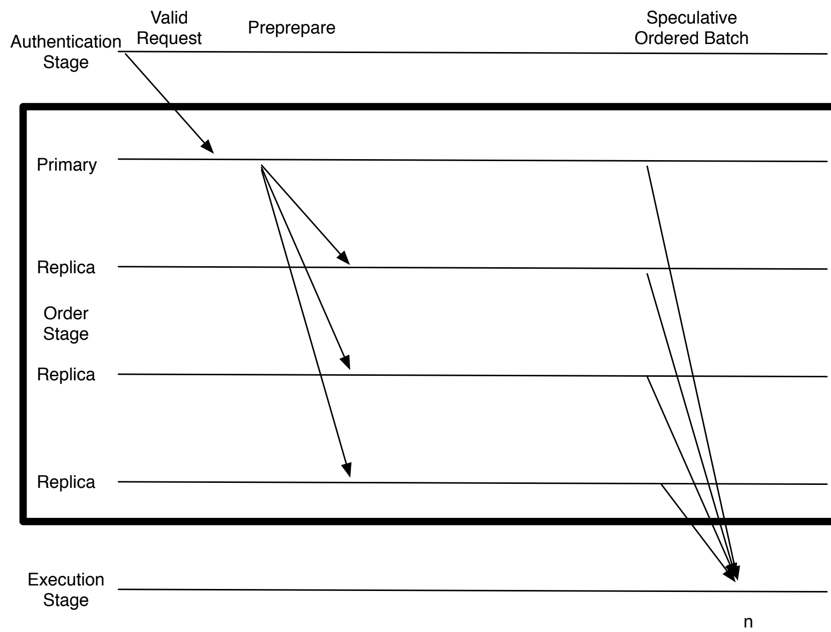

Figure 6.3: Basic communication pattern for speculative agreement.

history up to that batch. When the execution stage receives sufficient next batch messages to confirm batch n_o with history $\mathcal{H}$, it implicitly commits batch $n_o - 1$ with history $\mathcal{H}'$ provided that (a) $\mathcal{H}'$ is the immediate prefix of *history* and (b) the execution stage has received at least one next batch message (complete, tentative or speculative) for n'_o with history $\mathcal{H}'$.

Further refining failure counts. Several authors have noted [32, 56, 68, 92] that it is possible to provide speculative ordering even when failures occur. These systems introduce a new qualification to the UpRight goals: up despite at most u Byzantine failures, right despite at most r commission failures, *and fast despite at most* e *Byzantine failures.* We do not explore the specifics of fast ordering but observe that this work is complementary and can be incorporated into the order stage. Note that specifying fast failures exposes the true size of large quorums as $n - e$ and that a minimum of $\max\{2e + u + 2r + 1, 2u + r + 1\}$ acceptors are always sufficient for fast consensus[4]. Note that the protocols sketched above implicitly have $e = 0$.

Message authentication. We rely on MACs to authenticate all messages exchanged as part of the Zyzzyvark protocol.

Faulty client requests. Note that Zyzzyvark neither relies on signatures for client request authentication (Section 3.3) nor requires special handling for inconsistently authenticated client requests (Section **??**). We rely on the one-step transferable property of requests authenticated by the authentication stage to preemptively solve the problem.

Checkpoint management

As discussed in Section 5.3, the order stage maintains a base checkpoint, a secondary checkpoint, and a log of between CP_{interval} and $2 \times CP_{\text{interval}}$ batches ordered since the base checkpoint. The discussion in Section 5.3 focused on the definition of the order checkpoints and stage-level maintenance. In this section, we focus on how the order replicas coordinate to ensure that they each maintain a consistent order

[4]The familiar caveat that there exists specific configurations that require fewer acceptors applies.

checkpoint. Note that the checkpoint management discussed here is distinct from the checkpoint-operation to be discussed in Section 6.2.2

Before we get into the details of how Zyzzyvark replicas coordinate on order-stage checkpoints, it is important to note that checkpoint generation and garbage collection is a standard part of previous replication libraries such as PBFT [18], Zyzzyva [49], and Aardvark [24]. The checkpoint coordination in Zyzzyvark differs from its predecessors in two important ways. First, Zyzzyvark checkpointing (and by extension the order-stage checkpoints) are comparatively conservative: previous protocols ensure that each replica has one or two checkpoints and a log of at most $2 \times CP_{\text{interval}}$ batches since the oldest checkpoint, while Zyzzyvark guarantees that each replica always maintains two checkpoints and a log of between CP_{interval} and $2 \times CP_{\text{interval}}$ requests since the oldest checkpoint. Second, previous systems rely on a distinct protocol for checkpoint coordination while Zyzzyvark piggybacks checkpoint coordination onto normal operation.

Zyzzyvark piggybacks checkpoint coordination onto the agreement protocol that the system runs during normal-operation. The primary augments the PRE-PREPARE message for batch $(n_o+1) \mod CP_{\text{interval}} = 0$ with the order-stage checkpoint for $n_o - CP_{\text{interval}}$ and the replicas perform the traditional three phase agreement on this batch. The batch n_o is not ordered, i.e. the PRE-PREPARE containing n_o is neither sent by the primary nor processed by a replica, until the replica gathers a medium quorum of COMMIT messages for $n_o - 1$. Once $n_o - 1$ is committed, a replica can safely garbage collect checkpoint $n_o - 2 \times CP_{\text{interval}}$ and all batches $n'_o < n_o - CP_{\text{interval}}$. At the same time, the order replica generates a new checkpoint before considering the PRE-PREPARE message for batch n_o.

View change

Zyzzvyark, like PBFT and Zyzzyva, operates in "views." During a view v, replica $v \mod |replicaCount|$ is the designated primary. The view-change protocol is used to elect a new primary and determine the starting state for the next view $v+1$; in order for the system to remain consistent the new view must reflect all batches that were successfully ordered in the previous view. The Zyzzyvark view-change protocol uses standard techniques developed in PBFT [18] and Zyzzyva [49]. Zyzzyvark adopts the adaptive view-change triggers discussed in Chapter 3. Specifically, a replica

initiates a view-change when (a) the throughput in the current view drops below a constantly increasing threshold, (b) too much time passes between PRE-PREPARE messages, (c) the primary commits a detectable commission failure (e.g., attempts to include an invalid batch in a PRE-PREPARE message), or (d) a small quorum of other replicas initiate a view change.

6.2.2 Checkpoint-operation

Checkpoint-operation refers to the transfer of execution-stage checkpoints to and from the order stage and is conceptually distinct from the internal Zyzzyvark checkpointing discussed in the previous section.

Checkpoint-operation is conceptually simple: the execution stage proposes an execution-stage checkpoint to the order stage, the order stage accepts the checkpoint, and individual execution replicas learn the agreed upon checkpoint. Rather than implement another consensus protocol with the order replicas, we map checkpoint-operation onto the existing Zyzzyvark internal checkpoint mechanism.

We piggyback the checkpoint consensus protocol onto the Zyzzyvark checkpointing mechanism described above. The execution stage proposes an execution-stage checkpoint by sending a CP-TOKEN message containing the execution checkpoint to each order replica. The order replicas add the execution-stage checkpoint to the corresponding order-stage checkpoint. At designated points in the sequence of ordered batches, the primary includes the checkpoint in the PRE-PREPARE message and the checkpoint is subsequently agreed upon by the order replicas using the full three phase agreement path only if the checkpoint contained in the PRE-PREPARE message matches the checkpoint stored at each non-faulty order replica. Execution replicas can subsequently learn an execution-stage checkpoint after receiving a small quorum of $r + 1$ matching LOAD-CP messages from the order stage.

Note that because the execution stage consists of execution replicas (i.e. the proposer is the learners), the execution replicas generally only learn the value explicitly when recovering from a transient crash or catching up following an asynchronous interval. The communication pattern for the checkpoint consensus protocol is shown in Figure 6.4.

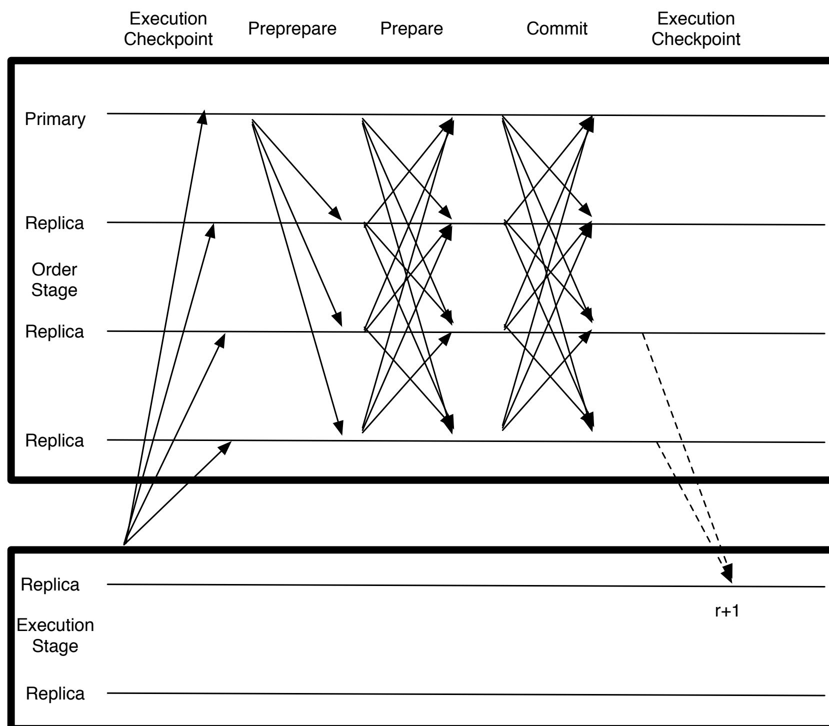

Figure 6.4: Basic communication pattern for the order stage checkpoint consensus protocol. Note that while the execution stage acts as a single proposer, each individual replica is a distinct learner. In the context of the UpRight library, learning is done only when a network or node failure occurs.

Message	Consensus Instance	Consensus Semantics
AUTH-REQ	normal	propose request
SPEC-BATCH TENT-BATCH COMP-BATCH	normal	learn ordered batch
CP-TOKEN	checkpoint	propose checkpoint
LOAD-CP	checkpoint	learn checkpoint
LAST-EXEC	both	utility — missed learning
RETRANSMIT	both	utility — should have learned

Table 6.2: Consensus semantics for messages related to the order stage. Each proposal or learn message is part of a single consensus instance. The utility messages are used by both consensus protocols.

6.2.3 Interactions with other stages

Replicating the order stage impacts how the authentication and execution stages process messages from the order stage and how they send messages to the order stage. To understand these changes, we must first put the intra-stage messages in the context of the normal and checkpoint consensus protocols.

There are a total of three intra-stage messages sent to the order stage and three-intra stage messages sent by the order stage. In Table 6.2 we divide these messages into three categories based on which consensus protocol the message is related to: normal-operation, checkpoint-operation, or both. The AUTH-REQ messages sent by the authentication stage are the proposals for normal-operation and the NEXT-BATCH messages sent to the execution stage are the corresponding learning messages. The CP-TOKEN messages sent by the execution stage are the proposals for checkpoint-operation and the load LOAD-CP messages sent to specific execution replicas are the corresponding learning messages. The RETRANSMIT messages sent to the execution stage and LAST-EXEC messages sent by individual execution replicas are utility messages indicating that something should have been learned or something was not learned respectively. The utility messages are used to ensure that an asynchronous network does not prevent the learners from learning accepted values.

The first time that the authentication stage authenticates client request n_c from client c, it sends an AUTH-REQ message containing request n_c to the current primary. On subsequent retransmissions of the request, the authentication stage sends the AUTH-REQ message to every order replica. Note that the first send results

in the request being ordered by the order stage (unless the primary is faulty or the network is ill-behaved) while subsequent sends trigger the retransmission process, notifying the execution stage that it (a) should retransmit any cached response for client c and/or (b) has missed one or more batches.

There are three distinct types of NEXT-BATCH messages sent to the execution replicas: speculative, tentative, and complete. An execution replica learns that a batch has been ordered only after gathering an appropriately-sized quorum of NEXT-BATCH messages: a large quorum of n speculative NEXT-BATCH messages, a medium quorum of $n - u$ tentative NEXT-BATCH messages, or a small quorum of $r + 1$ complete NEXT-BATCH messages. Execution replicas act on an ordered batch only once the batch has been learned, i.e., after receiving the appropriately sized quorum of matching NEXT-BATCH messages.

The execution stage sends CP-TOKEN messages to every order replica. Each order replica independently places the contained execution checkpoint in its local order checkpoint before participating in Zyzzyvark's checkpointing protocol. An execution replica learns that an execution checkpoint should be loaded when it receives a small quorum of $r + 1$ LOAD-CP messages. The small quorum is sufficient because checkpoint-operation relies on the three-phase-commit of the Zyzzyvark internal checkpoint mechanism.

The RETRANSMIT message is a hint that the order stage may have made accepted values that an execution replica has not yet learned. An execution replica acts on a RETRANSMIT message once it has received a small quorum of $r+1$ retransmission messages: enough to ensure that at least one correct order replica believes some action by the execution replica is necessary.

An execution replica sends LAST-EXEC messages to every order replica. The LAST-EXEC message explicitly states the last thing the sending replica learned and induces the order stage to resend the appropriate NEXT-BATCH and LOAD-CP messages to the execution replica.

6.2.4 Order stage properties

We identified a set of properties to be maintained by the order stage in Chapter 5. Before discussing how the replicated order stage fulfills those properties, we must first adjust the properties to account for replication. The replicated order stage is

correct if it is safe despite up to r commission failures and live despite up to u total failures. This results in a pair of simple modifications to the safety and liveness properties: the prefix "*if there are at most r commission failures, then*" is added to the safety properties and the prefix "*if there are at most u total failures and*" is added to the liveness properties.

We additionally further qualify OL1 to include the qualification "*sufficiently often during a sufficiently long synchronous interval.*" This additional qualification is made necessary by two properties of the Zyzzyvark protocol. First, for a batch to be ordered, Zyzzyvark requires coordination between multiple order replicas. The requisite communication is only guaranteed to happen during sufficiently long synchronous intervals. Second, Zyzzyvark relies on a primary to place requests in batches and propose an order for the batches. A faulty primary can fail to place specific requests into batches or fail to order requests entirely. The view-change protocol ensures that every primary is eventually replaced, guaranteeing that every request received infinitely often by the order stage during a synchronous interval is eventually received by a non-faulty primary and processed appropriately.

The augmented safety and liveness properties are presented below. Note that the augmentations are distinguished through *italics*.

OS1 *If there are at most r commission failures, then* only fetchable client requests authenticated by the authentication stage are placed into batches, and request n_c issued by client c is placed in at most one batch.

OS2 *If there are at most r commission failures, then* batches contain one or more requests and are assigned monotonically increasing batch identifiers n_o starting with 1 and increasing by 1 with each subsequent batch. For batches n_o and n'_o with associated times t and t', $n_o > n'_o \rightarrow t > t'$.

OS3 *If there are at most r commission failures and* request $n_c > 1$ issued by client c is in batch n_o, then request $n_c - 1$ issued by client c is in batch $n'_o < n_o$.

OS4 *If there are at most r commission failures, then* the stage always has stable checkpoint at n_o, where $n_o \% CP_{\text{interval}} = 0$, and $CP_{\text{interval}} \leq i \leq 2 \times CP_{\text{interval}}$ subsequent ordered batches.

OL1 *If there are at most u total failures and* the order stage receives*, sufficiently often during a sufficiently long synchronous interval,* unordered authenticated

request n_c issued by correct client c, then the order stage places the request in batch n_o and eventually sends a NEXT-BATCH message containing n_o to the execution stage.

OL2 *If there are at most u total failures and* the order stage receives an authenticated request n_c from client c that is already in batch n_o, then it instructs the execution stage to retransmit a response to request n'_c from client c in batch n'_o where $n'_c \geq n_c$ and $n'_o \geq n_o$.

OL3 *If there are at most u total failures and* (i) the execution stage requests all batches after n_e, (ii) the order stage has ordered batches through $n_o > n_e$, and (iii) $n_e + 1 \geq n_{\text{CP}}$, then the order stage resends all ordered batches from n_e through n_o.

OL4 *If there are at most u total failures and* the execution stage requests all batches after n_e and the order stage has ordered batches through $n_o > n_e$ and $n_e + 1 < n_{\text{CP}}$, then the order stage instructs the execution stage to load execution checkpoint n_{CP}.

The final point for consideration is how the replication strategy discussed in this section fulfills these properties. The safety properties OS1-4 describe the internal invariants maintained by Zyzzyvark and previous protocols such as PBFT [18], HQ [26], Zyzzyva [49], Aardvark [24], and others. Liveness property OL1 describes the basic liveness property of all asynchronous consensus protocols, and OL2-4 describe internal messages used as part of ensuring that every value proposed by a correct proposer is eventually learned by all correct learners.

Note that the "sufficiently often" condition of OL1 is satisfied through an interaction between correct clients and the authentication stage. Correct clients retransmit requests according to a regular schedule (at most four seconds between retransmissions) until a response to that request is received, and the authentication stage ensures that a request is retransmitted to the order stage at most once per four seconds. During synchronous intervals, the order stage receives an authenticated client request n_c issued by correct client c every four seconds until it is ordered.

6.3 Replicated execution stage

The primary responsibilities of the execution stage are delivering ordered batches to the application in the specified order and relaying the results of the executed requests to the clients. As part of processing each ordered batch, the execution stage notifies the authentication stage of the requests contained in that batch. Additionally, the execution stage sends an execution-stage checkpoint to the order stage every CP_{interval} batches. We view all of these activities as part of a single consensus protocol.

In this consensus protocol, the order stage acts as the single always correct proposer by proposing the sequence of ordered batches. The execution replicas accept the sequence of ordered batches and process batches in order. The clients, authentication stage, and order stage subsequently learn something—clients learn the results of executing batches in the specified ordered, the authentication stage learns which batch contains individual requests, and the order stage learns the execution-stage checkpoint. Note that each class of learners explicitly learns a subset of the information accepted by the acceptors; the portions of accepted state not learned explicitly are learned implicitly.

The handling of execution-stage checkpoints—specifically how the order stage learns the checkpoints—is the primary design decision that must be addressed when replicating the execution stage. We rely on indirect learning of the checkpoints, but note that other designs are possible.

The rest of this section details the design and replication requirements of the replicated execution stage. Section 6.3.1 describes the consensus protocol implemented by the execution stage in more detail. Section 6.3.2 describes alternate design options for handling execution-stage checkpoints. Section 6.3.3 describes the impact that relying on a replicated execution stage has on the authentication stage, the order stage, and clients. Section 6.3.4 describes how the replicated execution stage fulfills the properties of a correct execution stage described in Chapter 5.

6.3.1 Execution consensus

The consensus protocol implemented by the execution replicas is very simple and does not require any intra-stage communication because the order stage acts as a single always-correct proposer. Consensus with a single always-correct proposer

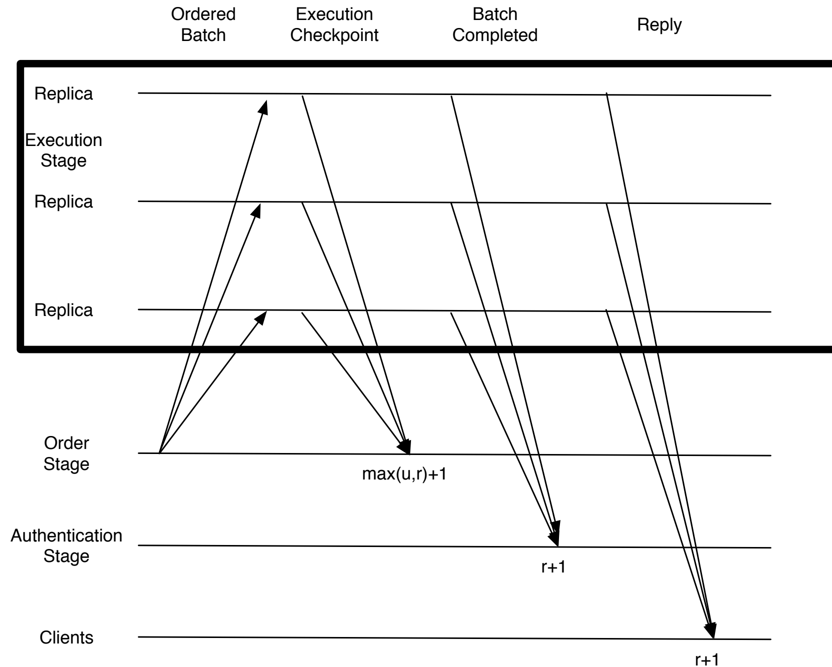

Figure 6.5: Execution consensus.

follows the communication pattern shown in Figure 6.5: the order stage proposes a batch of requests, the execution stage accepts the batch, and the clients learn the results of executing the batch/the authentication stage learns which batch each request is placed in/the order stage learns an execution checkpoint. As mentioned in Section 6.1, asynchronous consensus with a single correct proposer requires at least $u + r + 1$ replicas [56]. In this environment, learners can learn when they receive a quorum of $r + 1$ matching messages from the acceptors unless indirect learning requiring a quorum of $\max\{u, r\} + 1$ matching messages is necessary. The consensus protocol implemented by the execution replicas provides both regular learning (to the authentication stage and clients) and indirect learning (to the order stage).

Note that unlike the replicated order stage, the replicated execution stage does not require any coordination between internal replicas to implement consensus. The basic consensus protocol shown in Figure 6.5 can consequently be implemented by a set of execution replicas running the execution stage pseudo-code described in Section 5.7.4 without modification.

6.3.2 Execution-stage checkpoints

Recall that execution replicas send a hash of the execution-stage checkpoint to the order stage and not the checkpoint itself. When an execution replica falls behind or suffers from a transient crash, it learns the hash of the appropriate checkpoint to load and must subsequently fetch the checkpoint from another execution replica. Consequently, the order stage learns checkpoint hashes that are both correct and fetchable. This corresponds to indirect learning as discussed in Section 6.1 and requires at least $\max\{u, r\} + u + 1$ execution replicas. While the consensus protocol itself does not require any coordination between execution replicas, allowing individual execution replicas to fetch execution-stage checkpoints from another replica does require additional coordination.

Replica coordination

The only interaction required between execution replicas occurs when one replica falls far enough behind the other replicas that it must load a checkpoint that is not present locally. Figure 6.6 contains execution replica pseudo-code that handles the exchange of state between replicas. The additional messages introduced are shown in Table 6.3; full byte specifications of these messages can be found in Appendix A.3.

Recall from Section 5.3.2 that the execution stage loads a checkpoint upon receipt of a LOAD-CP message from the order stage. For replicas that have the specified checkpoint in their local storage (i.e. because they are recovering from a transient crash), loading the checkpoint is simple. However, it is also possible for a replica to receive the LOAD-CP checkpoint message and not have the execution stage checkpoint in local storage (e.g., because the replica became disconnected or suffered a transient crash and the other replicas made progress in its absence). When this occurs, the replica must first fetch the execution checkpoint described by the token $\mathcal{T}_{\text{cp}}$ contained in the load checkpoint message by sending an FETCH-EXEC-CP message to other execution replicas. Another execution replica responds with an EXEC-CP-STATE message containing the checkpoint state; the fetching replica compares the state to the checkpoint token compared in the LOAD-CP message and loads the state only if it is valid. As part of loading the execution checkpoint, the replica instructs the local copy of the application to load the application checkpoint contained in the execution checkpoint. If the application has the requisite state,

```
1   on rcv ⟨LOAD-CP, T_cp, n_o, o⟩_{μ_{o,e}} :
2       if ∃ exec_CP . n_e and hash(exec_CP . n_e) = T_cp then
3           CP_exec := exec_CP . n_e
4           state := app.loadCP(CP_exec.getAppCP())
5           if loadCP fails because application is missing state T_state then
6               send ⟨FETCH-STATE, T_state, e⟩_{μ⃗_{e,E}} to E
7       else
8           send ⟨FETCH-EXEC-CP, n, e⟩_{μ⃗_{e,E}} to E

10  on rcv ⟨FETCH-EXEC-CP, n, e⟩_{μ⃗_{e,E}} :
11      if checkpoint T_cp is locally available then
12          send ⟨EXEC-CP-STATE, n, S, this.e⟩_{μ_{this.e,e}}

14  on rcv ⟨EXEC-CP-STATE, n, S, e⟩_{μ_{e,this.e}} :
15      if requested checkpoint n and S matches T_cp then
16          CP_exec := exec_CP . n_e
17          state := app.loadCP(CP_exec.getAppCP())
18          if loadCP fails because application is missing state T_state then
19              send ⟨FETCH-STATE, T_state, e⟩_{μ⃗_{e,E}} to E

21  on rcv m = ⟨FETCH-STATE, T_state, e⟩_{μ⃗_{e,E}} :
22      S := app.getState(m.T_state)
23      send ⟨STATE, m.T_state, S, this.e⟩_{μ_{this.e,m.e}} to m.e

25  on rcv m = ⟨STATE, T_state, S, e⟩_{μ_{e,this.e}} :
26      state := app.loadState(m.S, m.T_state)
```

Figure 6.6: Execution replica pseudo-code related to intra-stage checkpoint and state transfer.

Message	Semantic meaning
FETCH-EXEC-CP	Fetch execution checkpoint n_e
EXEC-CP-STATE	Contains the state $\mathcal{S}$ of execution checkpoint n_e
FETCH-STATE	Fetch application state described by $\mathcal{T}_{\text{state}}$
STATE	Contains application state $\mathcal{S}$ described by $\mathcal{T}_{\text{state}}$

Table 6.3: State management messages exchanged between execution replicas.

then the checkpoint is loaded and operation can continue. It is likely, however, that the application may not have all of the requisite state available locally. If this is the case, the replica fetches the missing state by exchanging FETCH-STATE and STATE messages with other execution replicas.

The application can provide the full checkpoint to the execution stage or a token that describes the checkpoint concisely. If the former option is chosen then the execution replicas never fetch application state using the latter two messages in Table 6.3. If the latter option is chosen, then those two messages may be used to retrieve relevant state from other execution replicas.

Checkpoint alternatives

Note that we require $u + \max\{u, r\} + 1$ execution replicas rather than $u + r + 1$ because execution checkpoints are learned indirectly. Specifically, we require the execution stage to send a token, or cryptographic hash, describing the checkpoint to the order stage. Because we store the hash of the checkpoint at the order stage and the checkpoint at individual execution replicas, the order stage must be sure both that the checkpoint hash is correct and that the checkpoint is stored by at least one correct execution replica so that it can be fetched by another replica as needed.

There are two natural questions to ask. First, can we simplify the $\max\{u, r\}$ part of that expression? Second, can we avoid sending the execution-stage checkpoint (or its hash) to the order stage? At a high level, the answer to both questions is "no."

We can, in theory, simplify the $\max\{u, r\}$ portion of the expression to r by storing the entire execution-stage checkpoint (and not a hash) at the order-stage. We reject this approach because it can dramatically increase the network requirements of the system.

Similarly, we can remove the execution-stage checkpoint from the order stage entirely by increasing the number of execution replicas or relying on digital signatures to authenticate checkpoints. We reject these approaches for two reasons. First, to ensure that the order stage does not outrun the execution stage (i.e. orders several checkpoint intervals worth of batches that are not delivered to the execution stage because a lossy network), the execution stage must notify the order stage when it has completed a checkpoint. Second, augmenting that checkpoint notification to include a hash of the execution-stage checkpoint is less expensive than (a) authenticating execution-stage checkpoints with digital signatures or (b) increasing the number of execution replicas.

Can we simplify $\max\{u, r\}$**?** It is straightforward to simplify the required number of execution replicas to $r + u + 1$ by storing execution checkpoints at the order stage rather than tokens that describe the checkpoint. If this approach is taken, the order stage need only affirm that the checkpoint was correctly generated (i.e. receive at least $r + 1$ matching checkpoint messages) and does not need to ascertain that the checkpoint will be fetchable by another execution replica. The repercus-

sions of storing the full execution checkpoint at the order stage are very application and deployment dependent. In deployments where there are few clients and the application checkpoints are very small, then the execution checkpoints will be small and inexpensive to transfer to and store at the order stage. On the other hand, if there are large numbers of clients or the application checkpoints are large (gigabyte or even terabytes), then the costs of transferring the checkpoint from the execution replicas to the order stage and maintaining that checkpoint within the order stage may become prohibitive.

We choose to err on the side of conserving network bandwidth and simplifying the order stage and consequently store checkpoint hashes rather than the full checkpoints at the order stage.

Can we avoid sending the execution-stage checkpoint (or its hash) to the order stage? Previous work by on separating order from execution by Yin et al. [107] is based on a protocol where the order stage is oblivious to checkpoints generated by the execution stage and requires $u + r + 1$ execution replicas[5]. Lamport [60] presents a similar architecture that requires $u + 1$ execution replicas for a CFT system. We could adopt a similar approach and not store any reference to execution stage checkpoints at the order stage. Doing so would, however, require us to either use digital signatures[6] to authenticate execution stage checkpoints or rely on $u + 2r + 1$ execution replicas to provide one-step transferability within the execution stage. We believe it is better to store the execution-stage checkpoint at the order stage than to introduce digital signatures or increase the number of execution replicas.

In order to understand why digital signatures are necessary if there are only $u + \max\{u, r\} + 1$ execution replicas, let us consider a deployment where $u = r = 1$ and there are 3 execution replicas. Suppose replica a is correct, but does not receive any messages because of a lossy network. Meanwhile, the other two replicas, b and c, process ordered batches from the execution stage. Replica b is in fact Byzantine, but follows the protocol faithfully and generates correct client responses. After several checkpoint intervals, the network failure is repaired and replica a begins receiving

[5]Note that the work was presented as requiring $2f + 1$ execution replicas where $f = u = r$.

[6]Note that non-repudiation provided by digital signatures is equivalent to ∞-step transferability. Any authentication scheme that provides non-repudiation suffices.

messages again. At this point, a discovers that it is very far behind its peers and requests the most recent checkpoint from both b and c. Replica c responds with the correct checkpoint while replica b responds with a different checkpoint. Replica a is potentially in the unfortunate position of not being able to differentiate the correct checkpoint from a faulty checkpoint.

We could avoid this problem by having the replicas agree on the checkpoint and a proof that the checkpoint is correct. While this hypothetical proof would certainly ensure that only correct checkpoints are loaded, implementing the proof requires digital signatures (or another authentication scheme that provides non-repudiation) to ensure that the checkpoint will be loaded. Using digital signatures, b could gather a proof by waiting for digital signatures that match its checkpoint from r other replicas; this would require a total of at least $u + r + 1$ replicas.

Note that even with digital signatures, $u + \max\{u, r\} + 1$ execution replicas are required; $u + r + 1$ replicas do not suffice. Consider a setting where $u > r = 0$ and there are $u + 1 \geq 2$ total execution replicas. Assume, for the moment, that u replicas are caught behind a network partition resulting in only one execution replica processing batches from the order stage. The execution stage is guaranteed to be live despite up to u failures, so the system is able to continue processing requests as long as the clients continue to provide them. Note that the disconnected replicas are not actually faulty, but are prevented from receiving messages by an asynchronous and lossy network. Now suppose that the single active replica suffers a permanent crash and that the network failure is simultaneously repaired, but only after the system has processed several checkpoint intervals worth of requests. When the formerly disconnected replicas begin receiving messages again, they are unable to process the batches because their local state is not current, they do not possess a recent checkpoint because of garbage collection, and they are unable to fetch a recent checkpoint because the only replica that had the checkpoint is now failed. The net result is that the system cannot make safe progress despite the fact that no replica is guilty of a commission failure and only $1 \leq u$ replicas have failed.

We could replace digital signatures in the previous discussion with matrix signatures [3]. Doing so would trade the expense of digital signatures for additional execution replicas. Matrix signatures can be implemented using MACs, but require $3f = 2u+r+1$ replicas [3]; replacing digital signatures with matrix signatures would require $u + \max\{u, r\} + r + 1$ execution replicas.

Regardless of which approach we use to remove execution-stage checkpoints from the order stage, the execution stage must notify the order stage when it generates a checkpoint to prevent the order stage from outrunning the execution stage. Given this constraint, and the three options of (a) storing an execution-stage checkpoint token at the order stage, (b) using digital signatures to authenticate execution-stage checkpoints, and (c) increasing the number of execution replicas, we believe that storing an execution-stage checkpoint token at the order stage is the most reasonable decision.

Summary. Table 6.4 shows the tradeoffs for various checkpointing strategies: (1) the required execution replicas, (2) the network costs, and (3) the computation costs. We consider schemes that rely on digital signatures to have high computation costs and schemes that rely exclusively on MACs to have low computation costs. Schemes that push one or more copies of the execution checkpoint across the network have high network costs, while schemes that exclusively push hashes of the checkpoint have low network costs.

We compare four schemes for handling execution checkpoints. In the first scheme, we store a full checkpoint at the order stage. In the second scheme, we store the hash of the checkpoint at the order stage. In the third scheme, we do not store anything related to the checkpoint at the order stage and rely on digital signatures to generate a transferable proof for execution replicas to exchange with a valid checkpoint. In the final scheme, we replace digital signatures with matrix signatures. Storing the hash of the checkpoint at the order stage provides the right practical tradeoff between the required number of execution replicas and total network and computational costs.

6.3.3 Interactions with other stages

Replicating the execution stage impacts how the authentication stage, order stage, and clients process messages received from the execution stage and how they send messages to the execution stage. To understand these changes, we divide the inter-stage messages into two categories: consensus messages and state management messages. The consensus messages are the proposal and learning messages from the consensus protocol as well as the utility messages that alert the execution replicas

Checkpoint strategy	Required execution replicas	Network costs	Computation costs
Full CP at order stage	$u + r + 1$	high	low
Hash of CP at order stage	$u + \max\{u, r\} + 1$	low	low
Full CP at execution stage with digital signatures	$u + \max\{u, r\} + 1$	low	high
Full CP at execution stage with matrix signatures	$u + \max\{u, r\} + r + 1$	low	low

Table 6.4: Summary of replication requirements for different checkpoint storage strategies.

Message	Consensus Semantics
NEXT-BATCH	proposal
BATCH-COMPLETE	learn
REPLY	learn
CP-TOKEN	learn
RETRANSMIT	utility — learning failed
LAST-EXEC	utility — missed proposal

Table 6.5: Inter stage messages and their role in the execution consensus protocol.

that something should have happened. The state management messages are used to transfer request bodies from the authentication stage to the execution replicas.

Table 6.5 shows the six inter-stage messages that are part of the execution stage consensus protocol. The NEXT-BATCH message is the proposal and is sent by the order stage to all execution replicas. The BATCH-COMPLETE, REPLY, and CP-TOKEN messages are sent by execution replicas to the authentication stage, client that issued the request, and order stage respectively. Upon receipt of a quorum of $n - u$ matching messages, the recipient can safely learn the contents of the message. The RETRANSMIT message is sent by the order stage to every execution replica as a notification that either an accepted value was not learned or a proposed value was not accepted. An individual execution replica sends the LAST-EXEC message to the order stage to indicate that the replica did not receive a proposal—the order stage processes LAST-EXEC messages on a replica-by-replica basis and does not gather a quorum of matching messages.

Table 6.6 shows the two inter-stage state management messages. These messages are used to transfer request bodies from the authentication stage to individual

Message	Consensus Semantics
FETCH	none – state management
COMMAND	none – state management

Table 6.6: Inter stage messages related to stage management.

execution replicas. After receiving an ordered batch, an execution replica sends the FETCH message to the authentication stage indicating that the replica needs the specified request body. The authentication stage responds by sending a COMMAND message containing the request body to the execution replica that issued the FETCH message.

6.3.4 Execution stage properties

We identified the properties maintained by a correct execution stage in Chapter 5. A replicated execution stage is correct if it maintains the safety properties despite up to r commission failures and the liveness properties despite up to u total failures. Additionally, indirect learning requires **ES4** to hold despite up to u total failures. The requisite modifications to the safety and liveness properties are *italicized* below.

ES1 *If there are at most r commission failures, then* batch n_o is only delivered to the application if the last batch delivered to the application is $n_o - 1$.

ES2 *If there are at most r commission failures, then* only ordered batches are delivered to the application

ES3 *If there are at most r commission failures, then* only responses generated by the application are cached or sent to clients.

ES4 *If there are at most r commission failures and at most u total failures, then* execution stage maintains the execution checkpoint referenced by order base checkpoint in persistent memory.

ES5 *If there are at most r commission failures, then* then the execution stage provides deterministic and replayable execution of ordered batches.

EL1A *If there are at most u total failures and* the execution stage receives ordered batch n_o and the last batch it has delivered to the application is $n'_o < n_o$, then

it fetches the request bodies for requests in batch n_o from the authentication stage and notifies the authentication stage that the contained requests have been ordered.

EL1B *If there are at most u total failures and* the execution stage has all of the request bodies for batch n_o and the last batch it delivered to the application is $n_o - 1$, then the execution stage delivers batch n_o to the application.

EL2 *If there are at most u total failures and* the execution stage receives a response from the application, then it stores the response for retransmission and sends the response to the responsible client.

EL3 *If there are at most u total failures and* the execution stage receives a retransmission instruction for request n_c from c in batch n_o and the last batch executed by the execution stage is $n_e > n_o$, then the execution stage resends the response to the most recent request $n'_c \geq n_c$ executed for client c and notifies the authentication stage that n'_c has been ordered no later than batch n_e.

EL4 *If there are at most u total failures and* the execution stage receives a retransmission instruction for request n_c from client c in batch n_o and the last batch executed by the execution stage is $n_e < n_o$, then the execution stage informs the order stage that it has missed the batches since n_e.

EL5 *If there are at most u total failures and* the execution stage receives an instruction to load checkpoint n_e from the order stage, then it loads execution checkpoint n_e.

Each correct execution replica independently implements the safety and liveness properties above. Coordinating the replicas through the consensus protocol as discussed in Section 6.3.1 ensures that a collection of at least $\max\{u, r\} + u + 1$ execution replicas is sufficient to implement a correct execution stage.

6.4 Replicating authentication stage

The authentication stage is responsible for authenticating requests issued by authorized clients, caching the body of those requests, and delivering a hash of authen-

ticated requests to the order stage. This process maps to a collection of consensus protocols, one per client per request. The consensus protocols share the same set of acceptors (the authentication replicas) and learners (the order stage) and are differentiated by the proposer (each client is a proposer in a distinct instance of consensus each time it issues a distinct request).

The rest of this section details the design of the replicated authentication stage. Section 6.4.1 describes the implementation of each authentication replica and intra-stage coordination. Section 6.4.2 describes the impact of replicating the authentication stage on clients, the order stage, and the authentication stage. Section 6.4.3 describes how the replicated authentication stage fulfills the properties of a correct authentication stage described in Chapter 5.

6.4.1 Authentication consensus

We map the authentication stage to the acceptors in a collection of consensus protocols. Each distinct request issued by a client c is the proposal for a distinct instance of consensus. The authentication replicas accept the request. The order stage learns request hashes that (a) correspond to requests issued by authorized clients, (b) correspond to request bodies that are cached by the authentication stage, and (c) are one-step transferable.

As discussed in Section 6.1, a total of $u+r+1$ replicas are sufficient to provide basic consensus with a single proposer and satisfy requirement (a). Requirement (b) lays out the need for indirect learning and a baseline of $u + \max\{u, r\} + 1$ replicas. Requirement (c) requires one-step transferability and increases the requisite number of authentication replicas to the final total of $u + \max\{u, r\} + r + 1$.

Authentication replicas implement the authentication stage pseudo-code presented in Chapter **??** and do not communicate with each other when processing client requests. The communication induced by this (lack of) coordination is similar to the consensus protocol employed by the execution stage and can be found in Figure 6.7. The authentication stage differs from the execution stage in two important ways.

First, the authentication stage does not require any checkpoints to be coordinated between the authentication replicas because it is okay for replica state to diverge. Values learned from the execution stage depend on each other—it is impossible for the execution stage to process batch n_o without first processing batch

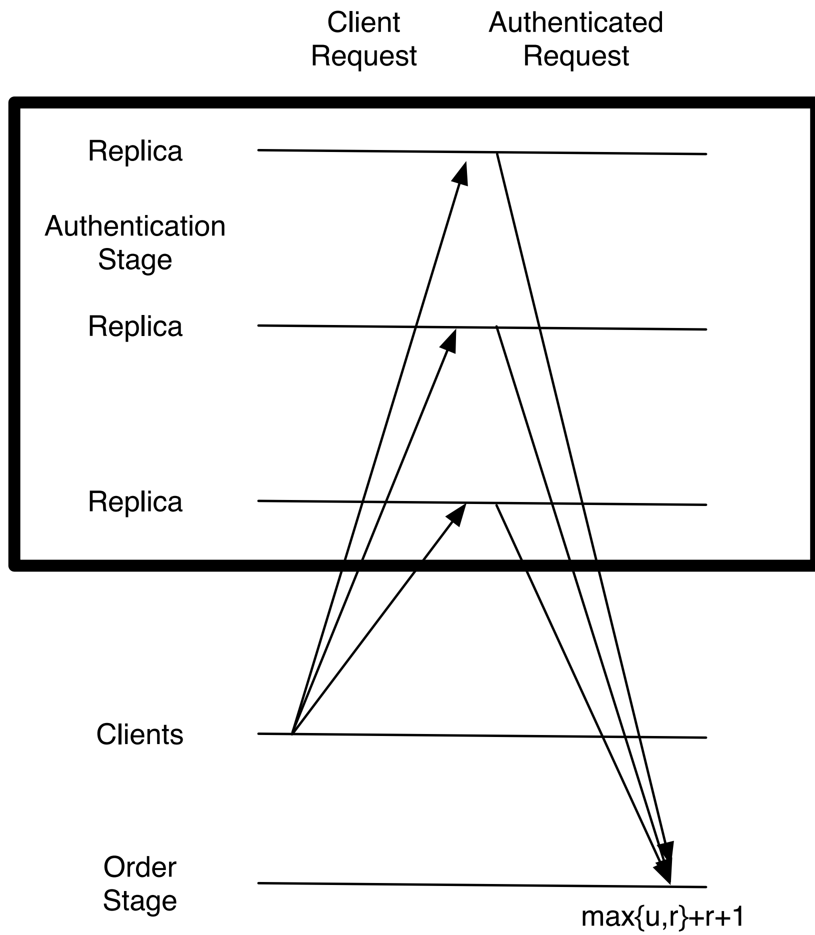

Figure 6.7: Authentication consensus.

Message	Sent by
$\langle$CLIENT-REQ, $\langle$REQ-CORE, c, n_c, OP$\rangle, c\rangle_{\vec{\mu}_{c,\mathcal{F}}}$	client
$\langle$AUTH-REQ, $\langle$REQ-CORE, c, n_c, hash(OP)$\rangle_{\vec{\mu}_{f,\mathcal{O}}}, f\rangle_{\vec{\mu}_{f,\mathcal{O}}}$	authentication stage
$\langle$BATCH-COMPLETE, $v, n_o, \mathcal{C}, e\rangle_{\vec{\mu}_{e,\mathcal{F}}}$	execution stage
$\langle$FETCH, n_o, c, n_c, hash(OP)$, e\rangle_{\vec{\mu}_{e,\mathcal{F}}}$	execution stage
$\langle$COMMAND, n_o, c, n_c, OP$, f\rangle_{\mu_{f,e}}$	authentication stage

Table 6.7: Messages sent to and from the authentication stage.

$n'_o < n_o$. Values learned from the authentication stage, on the other hand, are independent of each other—learning that "client c said X" does not require any knowledge that "client c' said Y."

Second, the authentication stage is required to provide one-step transferability of authenticated requests. This requirement is important because the order stage is based on a primary-led consensus protocol, and we rely on MACs for message authentication.

6.4.2 Interactions with other stages

The authentication stage receives three messages from other stages and sends two messages to other stages. The complete set of messages sent to and processed by the authentication stage is shown in Table 6.7.

The first time a client c issues request n_c, the client optimistically assumes that the network is well-behaved and there are no failed authentication replicas and sends a CLIENT-REQ message to a preferred medium quorum of $n-u$ authentication replicas. The preferred quorum used by client c consists of the $n-u$ authentication replicas starting with replica $c \mod n$. If c retransmits the CLIENT-REQ message containing request n_c then it sends the request to all authentication replicas on the assumption that either the network is ill-behaved or one or more replicas in its preferred quorum are in fact faulty.

The order stage primary gathers a medium quorum of $\max\{u, r\} + r + 1$ AUTH-REQ messages before placing a request in a batch. Order replicas, including the primary, gather a small quorum of $r + 1$ AUTH-REQ messages before sending a RETRANSMIT message for a previously ordered request.

The execution stage sends BATCH-COMPLETE messages to all authentication replicas. Execution replicas initially send FETCH messages to a specific member

of each client's designated preferred quorum—authentication replica $c \mod n$. If the execution replica does not receive the request body, then it resends the FETCH message to all authentication replicas. An execution replica may act on the first FETCH message that it receives from an authentication replica, though it checks the body against the request hash contained in the ordered batch before acting on the body.

6.4.3 Authentication stage properties

We modify the authentication stage properties identified Chapter 5 to accommodate the UpRight design goals. A replicated execution stage is correct if it maintains the safety properties despite up to r commission failures and the liveness properties despite up to u total failures. The requisite modifications to the safety and liveness properties are *italicized* below.

AS1 *If there are at most r commission failures, then* only requests issued by authorized clients are authenticated and every authenticated request is one-step transferable.

AS2 *If there are at most r commission failures and at most u total failures, then* every authenticated request referenced by a batch ordered since the base checkpoint at the order stage or not yet ordered is fetchable.

AL1a *If there are at most u total failures and* the authentication stage receives a request n_c issued by correct client c and there is no pending request $n''_c < n_c$, then request n_c is authenticated and sent to the order stage.

AL1b *If there are at most u total failures and* the authentication stage receives a request n_c issued by correct client c and there is a pending request n'_c, then request n'_c is authenticated and sent to the order stage.

AL2 *If there are at most u total failures and* the authentication stage receives a fetch body message from the execution stage for a authenticated request n_c issued by client c, then the authentication stage responds with the request body.

The safety properties AS3 and AS4 are maintained by individual authentication replicas and are not properties maintained by the authentication stage as a whole.

Note that in the context of the end-to-end system AS3 and AS4 are not strictly necessary. These two properties are used to limit the rate at which faulty clients can force the system to consume storage and bandwidth.

AS3 At most one request per identifier n_c per authorized client c is authenticated.

AS4 When request n_c from client c is authenticated, no request $n'_c > n_c$ has been authenticated and there is no pending request $n'_c < n_c$.

Each authentication replica implements the protocol described in Figure 5.7 and correct replicas maintain local versions of the authentication stage safety and liveness properties (intuitively, replace "authentication stage" with "authentication replica" and ignore the failure count qualifier). The union of replicas that individually provide the specified properties ensures that the stage as a whole provides the properties.

6.5 Implementation and performance

We implement the UpRight library based on the inter-stage protocol described in Chapter 5 and the replicated stages previously discussed in this chapter in Java and regrettably must name the prototype JSZyzzyvark[7]; J-Zyzzyvark refers to a configuration where we omit writing to disk to compare more meaningfully our prototype with prior Byzantine agreement protocols and to expose bottlenecks in our protocol. We believe that a Java-based solution is more suitable for widespread deployment with the Java-based Zookeeper and HDFS systems than a C implementation despite the difference in performance between C and Java implementations. We also note that logging actions to disk places a ceiling on throughput so the benefits of further optimization may be limited.

We run our servers on 3GHz dual-core Pentium-IV machines, each running Linux 2.6 and Sun's Java 1.6 JVM. We use the FlexiProvider [36] cryptographic libraries for MACs and digital signatures and the Netty [74] networking library for asynchronous Java I/O. Nodes have 2GB of memory and are connected via a 100Mbit/s Ethernet. Except where noted, we use separate machines for authentication, order, and execution replicas.

[7] "J" because the prototype is implemented in Java, "S" because the prototype stores state to stable storage, and "Zyzzyvark" because the prototype is based on the Zyzzyvark protocol.

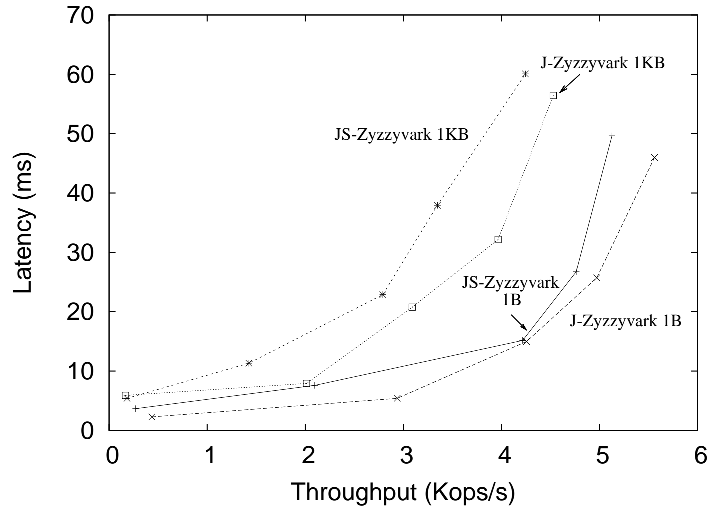

Figure 6.8: Latency v. throughput for J-Zyzzyvark and JSZyzzyvark.

The UpRight library (client, authentication, order, and execution stages) comprise 20,403 lines of code (LOC).

Method. Our basic experimental setup involves correct clients that operate in a closed loop—that is they issue requests one at a time and do not issue request i until they receive a response to request $i-1$. Unless otherwise noted, correct clients issue 100k requests. We increase system load by increasing the number of clients. Clients record the time at which each request is issued and the response received. We calculate the average latency of all requests issued by all clients. We calculate per second throughput by dividing the total duration of the experiment, in seconds, by the total number of requests issued by all clients. Each data point corresponds to a single experimental run.

Response time and throughput. Figure 6.8 shows the throughput and response time of J-Zyzzyvark and JSZyzzyvark. We vary the number of clients issuing 1 byte or 1 KB null requests that produce 1 byte or 1 KB responses and drive the system to saturation. We configure the system to tolerate 1 fault ($u = r = 1$).

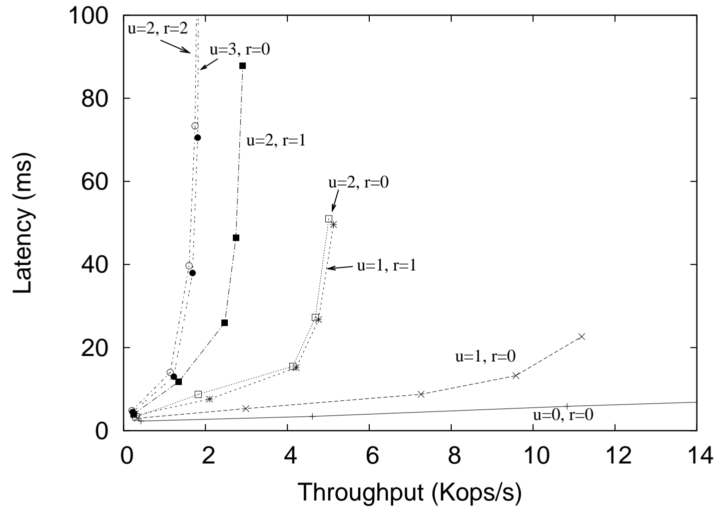

Figure 6.9: Latency v. throughput for JSZyzzyvark configured for various values of r and u.

For small requests J-Zyzzyvark's and JSZyzzyvark's peak throughputs are a respectable 5.5 and 5.1 Kops/second, which suffices for our applications. They are comparable to unmodified Zookeeper's peak throughput for small read/write requests, and they appear sufficient to support an HDFS installation with a few thousand active clients. Peak throughputs fall to 4.5 and 4.2 Kops/second for a workload with larger 1KB requests and 1KB replies.

For comparison, in Chapter 3 we reported small request throughputs of 7.6, 23.8, 38.6, 61.7, and 66.0 Kops/s for the C/C++-based HQ [26], Q/U [1], Aardvark [24], PBFT [18], and Zyzzyva [49] on the same hardware. For environments where performance is more important than portability or easy packaging with existing Java code bases, we believe a well-tuned C implementation of Zyzzyvark with writes to stable storage omitted would have throughput between that of Aardvark and Zyzzyva—our request validation and agreement protocols are cheaper than Aardvark's, but our request validation is more expensive than Zyzzyva's.

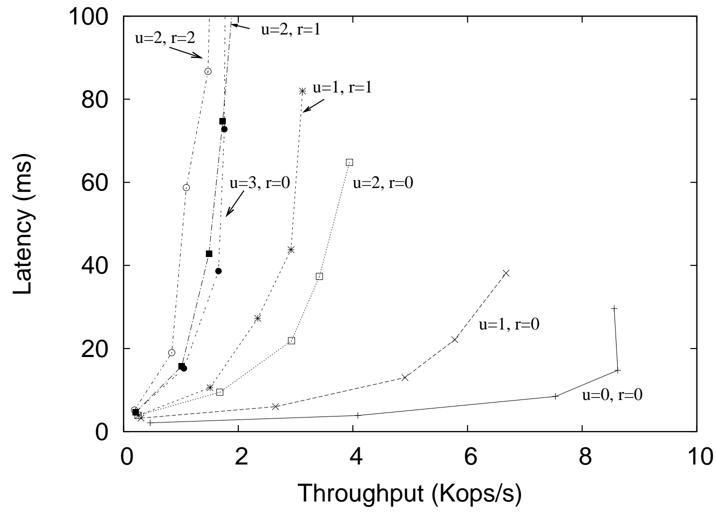

Figure 6.10: Latency v. throughput for JSZyzzyvark configured for various values of r and u with authentication, order, and execution replicas colocated.

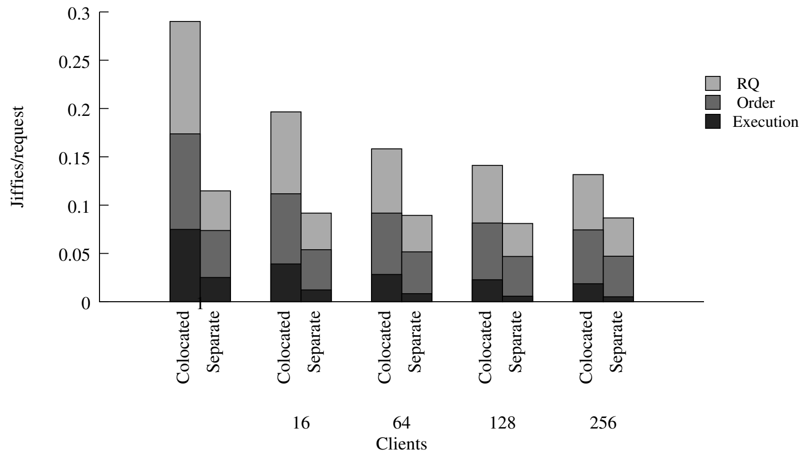

Figure 6.11: Jiffies per request. RQ indicates the jiffies at the authentication stage; Order indicates the jiffies at the order stage; Execution indicates the jiffies at the execution stage.

Other configurations. Figure 6.9 shows small-request performance as we vary u and r. Recall that Zyzzyvark requires $2u + r + 1$ authentication and order replicas and $u + r + 1$ execution replicas to ensure that it can tolerate u failures and remain up and r failures and remain right. Peak throughput is 11.1 Kops/second when JSZyzzyvark is configured with $u = 1$ and $r = 0$ to tolerate a single omission failure (e.g., one crashed replica), and throughput falls as the number of faults tolerated increases. For reference, we include the $u = 0$ $r = 0$ line for which the system has just one authentication, order, and execution replica and cannot tolerate any faults; peak throughput exceeds 22 Kops/s, at which point we are limited by the load that our clients can generate.

Figure 6.10 shows small-request performance when the authentication, order, and execution replicas are co-located on $2u + r + 1$ total machines. Splitting phases across machines improves peak throughput by factors from 1.67 to 1.04 over such co-location when any fault tolerance is enabled, with the difference falling as the degree of fault tolerance increases. Figure 6.11 shows the the number of CPU jiffies (4ms of CPU time) per request summed across authentication, order, and execution processes on all replicas for two configurations: (1) when all stages share a common set of machines and (2) when each stage runs on its own separate set of machines. As load increases, larger batch sizes amortize some costs, reducing processing per request. In the second configuration, the bottleneck is the order stage, and the execution replicas are lightly utilized. The higher per-request processing cost that we observe in the first configuration is unexpected and we have not to date identified a convincing explanation for it.

Request authentication. In Figure 6.12 we examine the throughput of the JSZyzzyvark prototype configured for $u = 1$ and $r = 1$ and using different strategies for client request authentication. The *MAC RQ* line shows performance of the default JSZyzzyvark configuration that relies on MAC-based matrix signatures formed at the authentication stage. In contrast, the *SIG no RQ* line omits the authentication stage entirely and shows the significant performance penalty imposed by relying on traditional digital signatures for request authentication, as in Aardvark. The *MAC no RQ* line shows the performance that is possible in a system that relies, like PBFT, on MAC authenticators and uses no authentication stage for client authentication. In a system where the robustness risk and corner-case complexity of

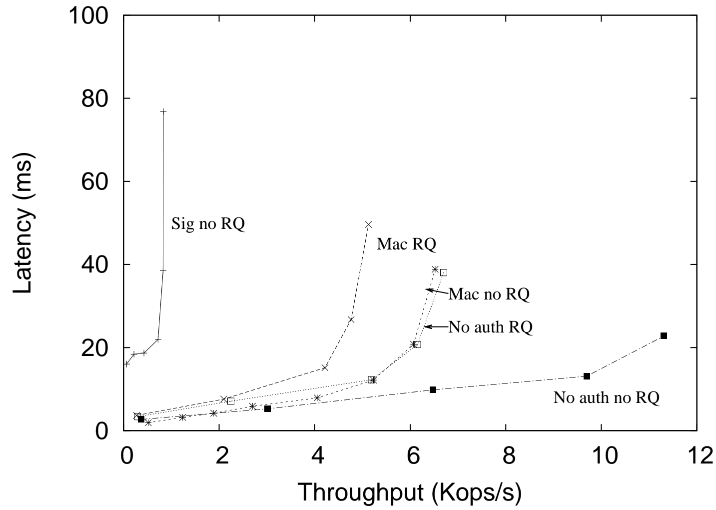

Figure 6.12: JSZyzzyvark performance when using the authentication replica and matrix signatures, standard signatures, and MAC authenticators. (1B requests)

relying on MAC authenticators as opposed to matrix signatures are viewed as acceptable, this configuration may be attractive. For comparison, the *no auth RQ* line shows performance when we use the authentication stage but turn off calculation and verification of MACs, and the *no auth no RQ* line shows performance when we eliminate the authentication stage and also turn off calculation and verification of MACs.

Request digests. Figure 6.13 demonstrates the value of storing requests at the authentication stage so that the order stage can operate on digests rather than full requests. We configure the system for $u = 1$ and $r = 1$. For small requests (under 64 bytes in our prototype), the authentication stage sends full requests and the order replicas operate on full requests; the figure's *1B Request* line shows performance for 1 byte requests. The *1KB Digest* and *10KB Digest* lines show performance for 1KB and 10KB requests when authentication replicas store requests and send request digests for ordering, and the *1KB Request* and *10KB Request* lines show performance with the request storage and digests turned off so that order replicas

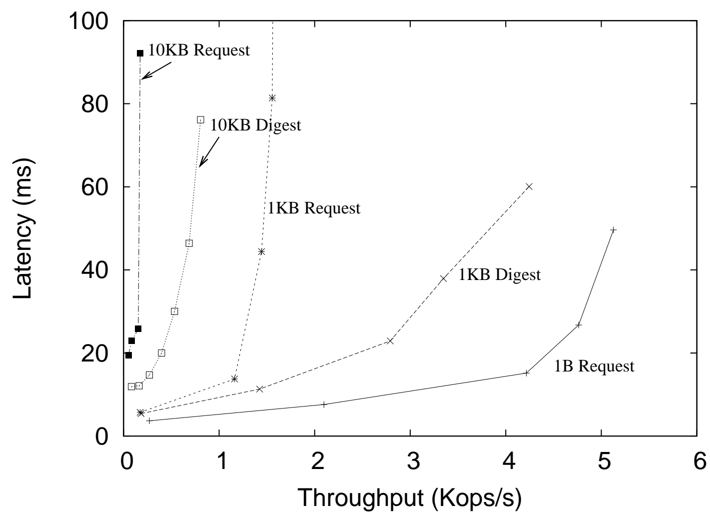

Figure 6.13: JSZyzzyvark performance for 1B, 1KB, and 10KB requests, and for 1KB and 10KB requests where full requests, rather than digests, are routed through order replicas.

operate on full requests. Storing requests at the authentication more than doubles peak throughput for 1KB and 10KB requests.

6.6 Discussion

Separating the stages in the UpRight architecture facilitates a clean and modular design and implementation. The separation is a logical separation only and there is no fundamental reason to not colocate an authentication, order, and execution replica on the same machine. When this is done, the replicas communicate as if they are all on distinct machines[8]. It is tempting to bind colocated replicas to each other more tightly in order to eliminate the intra-stage communication steps, especially the all-to-one authentication-to-order and all-to-all order-to-execution steps. This temptation is misguided, however, as these communication steps are intrinsic to important properties of our design. The authentication-to-order step allows us to order request hashes rather than full request bodies and avoid the dangers of inconsistently authenticated client requests without relying on digital signatures while the order-to-execution step allows us to ensure that no replica is ever required to roll back application state.

The authentication stage is responsible for authenticating requests issued by clients and caching the bodies of those requests until they are ordered. The former responsibility simplifies the protocol for agreeing on the order of batches by ensuring that any request deemed valid by a correct primary will also be deemed valid by a correct replica (discussed in Section 3.3 and Section 6.2) while the latter responsibility reduces the network bandwidth required when agreeing on the batch execution order. Removing the all-to-one communication between the authentication stage and the order stage primary effectively eliminates the authentication stage from the architecture and would require the agreement protocol to order client requests, increasing inter-order-stage bandwidth and storage costs, and either use digital signatures to authenticate client requests or implement the code to handle the corner case where a client request is inconsistently authenticated (or suffer the consequences observed in Section 3.6.2).

The all-to-all communication between the order and execution stages ensures

[8]An obvious optimization in that case is to send messages to colocated replicas using memory channels rather than relying on the network infrastructure.

that the execution replicas only execute a batch of requests when that batch is definitely the next batch in the sequence. One important side effect of this design is that the execution replicas are *never* required to roll application state back. While rolling application state back is feasible, the standard mechanism used to roll application state back is to load an old checkpoint and then replay the requests since the old checkpoint. This can be an expensive activity and is one that should be avoided. In order to maintain this property of no application roll back with the order and execution stages merged, the replicas could not execute batches until after the PREPARE phase completed—merging the order and execution stages would not reduce the required number of message delays and could actually increase the total network traffic since there are always at least as many order replicas as execution replicas. Of course, if an application can support fine grained checkpoints and roll back then it may be tenable to allow the execution replicas to execute batches speculatively, rather than rely on speculation simply to speed the learning process.

6.7 Conclusion

This chapter describes the replicated implementation of the authentication, order, and execution stages specified in Chapter 5. The key to understanding the design of each stage is understanding the problem of consensus and the impact that the context that consensus is being solved in has on the number of required replicas. The authentication, order, and execution stages each implement the acceptors for one or more instances of consensus. Once those instances are identified, the design of each stage is straightforward.

Understanding the mapping between each stage and consensus allows us to implement each stage with a minimal number of replicas and also to identify and fix problems with previous systems that attempt to separate the stages of state machine replication.

In the context of this chapter, UpRight fault tolerance is a fact of the implementation and design. The next chapter describes our experience incorporating the UpRight library into HDFS and ZooKeeper. The value of the flexibility of UpRight fault tolerance will be discussed in that context.

Chapter 7

UpRight Applications

The previous chapters have focused on the challenges of implementing the library specification provided in Chapter 4. This chapter, in contrast, focuses on our experiences adapting the ZooKeeper distributed coordination service [108] and Hadoop distributed file system (HDFS) [43] to be compatible with the UpRight library. Our goal in this chapter is to use these two systems as case studies to demonstrate three points:

1. The application changes required to make existing applications UpRight are small in scope and complexity.

2. UpRight applications provide flexible fault tolerance—a single code base can provide different levels of crash and Byzantine fault tolerance through simple modifications to a configuration file.

3. The performance of UpRight applications is competitive with the performance of the unmodified code bases.

While the second point can be objectively demonstrated, the first and third are largely subjective and are demonstrated through observation and experience reports. While the experiences reported in this chapter are specific to our two case studies, we believe the lessons learned are applicable to other applications for which UpRight is likely to be of interest.

Recall that we specified a set of properties to be maintained by the application in Chapter 4. Intuitive statements of these properties are shown in Table 7.1 for

APPS1	Only requests contained in batches received from the library are executed.
APPS2	Requests are executed deterministically
APPS3	Checkpoints are generated deterministically
APPS4	Loading a checkpoint puts the application into the state it was in when the checkpoint was generated.
APPL1	The function call $execute(batch)$ returns.
APPL2	The function call $takeCP()$ returns.
APPL3	The function call $loadCP()$ returns.

Table 7.1: Informal statement of application requirements.

easy reference. The primary challenge we face in this chapter is adapting ZooKeeper and HDFS to meet these requirements, namely to provide deterministic execution (APPS2) and on-demand deterministic checkpoint generation (APPS3,4). The UpRight library, because it is implemented in Java, implicitly requires applications to be written in Java though there is no fundamental reason that the library cannot be ported to support applications written in other programming languages.

Application requirement APPS1 requires the application to process only valid requests, that is requests delivered to the application by the by the replication library, and is trivial to maintain.

Application requirement APPS2 requires the application to execute batches deterministically—given an application state and a batch of requests, the application should always produce the same set of responses and end in the same application state. We employ standard techniques for ensuring deterministic request execution [1, 18, 24, 26, 49, 50, 86, 92, 104, 107] when modifying ZooKeeper and HDFS to fulfill this requirement.

Application requirements APPS3 and APPS4 require the application to produce determinist checkpoints on demand. As discussed at the end of Section 5.3.2, generating application checkpoints on demand plays an important role in bounding the state stored by the UpRight library and bringing "slow" replicas up to speed. We suspect that applications for which replication is appropriate already rely on some form of checkpointing to handle power outages and other transient failures; our experiences with ZooKeeper and HDFS reinforce this belief. However, our experiences indicate two challenges to providing deterministic on demand checkpoints as required by the UpRight library. First, the checkpoints are not always deterministic

or complete—HDFS stores some required information as soft state and relies on an asynchronous protocol to replenish that state following a transient crash while ZooKeeper creates fuzzy snapshots that are equivalent but not identical. Second, applications are generally tuned to generate checkpoints at a much slower rate (10,000s of requests) than the UpRight library (100s of requests). To address the first challenge, we modify the native HDFS and ZooKeeper to provide complete and deterministic checkpoints on demand. We address the second challenge by implementing a generic log-based checkpointing mechanism that combines large-granularity application checkpoints with a batch log to produce the fine-grained checkpoints required by the UpRight library.

Note that the requirement that the application generate deterministic checkpoints on demand departs from the application requirements imposed by most previous libraries [1, 18, 24, 26, 49, 50, 86, 92, 104, 107]. These libraries do not expose checkpointing to the application and instead require the application developer to store relevant application state in a memory space managed by the replication library. The replication library, in turn, handles checkpoint generation and rollback without any application support. The primary drawback to this approach is that it can require significant parts of the application to be rewritten. In contrast, the requirement that the application implement its own checkpointing actually facilitates reuse of existing functionality.

We approach the challenges of modifying an application using the framework shown in Figure 7.1. From the application developer's perspective, the UpRight library is a black box. The application developer's sole responsibility is attaching the application to the library—the application server must attach to the execution-stage and each application client attaches to a distinct library client instance. We conceptually divide the execution stage and client into three distinct components to facilitate this process:

1. A generic *shim*: the shim moderates communication between the stages in the UpRight architecture. The shim implements the execution stage of the UpRight architecture and exports a simple API to the application.

2. *Application-specific glue*: The application-specific glue is the bridge between the UpRight library and the application. it is the one part of the system where knowledge of both how the application works and awareness of repli-

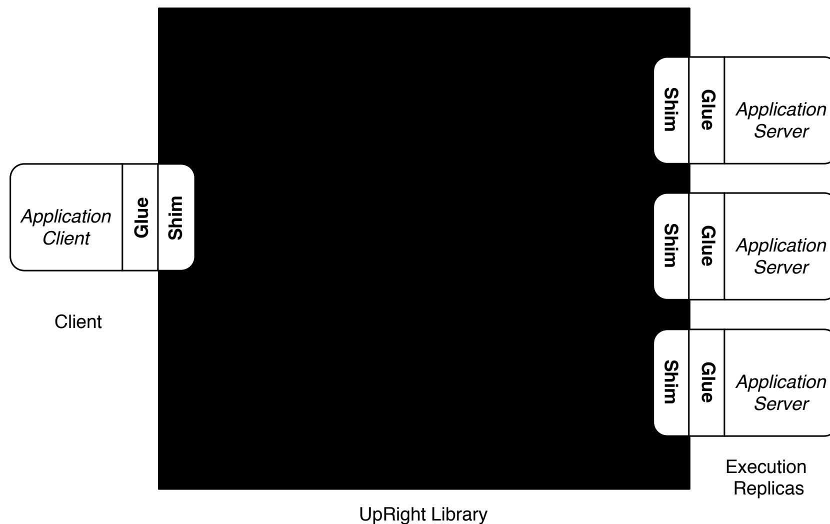

Figure 7.1: UpRight application architecture from an application developer perspective. The UpRight library is a black box with a well defined interface. At both the client and the server, the developer implements application-specific glue that connects the library shim to the original application.

cation are mixed. The glue contains the application-specific knowledge necessary for replication: demuxing request batches, maintaining and constructing application-specific instantaneous checkpoints, identifying which state must be transferred in order to load a checkpoint, etc. At the client stage the glue performs appropriate request pre-processing and response post-processing.

3. The *application*: the application is the (mostly) unmodified application. The application is responsible for providing deterministic execution and deterministic checkpoints. Our goal is to keep changes to the application to a minimum and isolate the application awareness of replication in the glue.

The rest of this chapter contains six sections. Section 7.1 provides an overview of what is needed to provide deterministic request execution that satisfy APPS1, APPS2, and APPL1. Section 7.2 describes a generic checkpoint management scheme that we adapt for use with both ZooKeeper and HDFS to provide APPS3, APPS4, APPL2, and APPL3. Section 7.3 describes specifics of our experience with HDFS and reports on observed performance. Section 7.4 describes specifics of our experience

with ZooKeeper and reports on observed performance. Section 7.5 summarizes our experiences with ZooKeeper and HDFS and highlights the key lessons learned.

The Java APIs for the client and server shims and glues can be found in Appendix B.

7.1 Request Processing

We identified three main challenges in ensuring that batch execution meet the requirements of APPS1 and APPS2. APPS1, unsurprisingly, was straightforward. Enforcing APPS2 required us to (a) demux batches of requests into individual requests, (b) handle sources of nondeterminism including PRNG seeds and system time, and (c) address challenges associated with multi-threading. We report on the techniques we found sufficient in our work with HDFS and ZooKeeper. Note that the experiences we report here and in the rest of this chapter are pragmatic responses to the challenges that we encountered and are not driven by first principles. Nonetheless, we believe that these challenges (and our solutions) are likely to be relevant to many replicated applications.

Our core strategy to providing deterministic execution is ensuring that our applications execute requests deterministically and sequentially based on the order they appear in the batch. Ensuring deterministic execution is a well explored research area [48, 73]. Our approach to this problem is guided by the goal of making minimal changes to the application and the run-time, rather than identifying a principled and potentially invasive approach that can automatically be applied to an arbitrary application.

Demuxing batches. The UpRight library provides the application with batches of requests. The glue/application must demux the batches into individual requests for execution. We take the simple approach of interpreting the batch sequentially—the first request is executed first, the second is executed second, and so forth.

Nondeterminism. Many applications rely on real time or random numbers as part of normal operation. These factors can be used in many ways including garbage collecting soft state, naming new data structures, or declaring uncommunicative nodes dead. Each request issued by the UpRight shim to the application server glue

is accompanied by a time and random seed to be used in conjunction with executing the request [18]. UpRight applications must be modified to rely on these specified times rather than the local machine time and to use the random seed as appropriate when using a pseudo random number generator.

Multithreading. Parallel execution allows applications to take advantage of hardware resources, but application servers must ensure that the actual execution is equivalent to executing the request batches sequentially in the order specified by the UpRight library. The simplest way to enforce this requirement is for the glue to complete execution of batch n_o before the execution of batch $n_o + 1$ and request i of batch n_o before beginning execution of request $i + 1$. Although we take the simple approach of executing batches and requests sequentially, more sophisticated glue may process the requests of an individual batch in parallel [50, 100] or may even support parallel execution of batches as long as all replicas generate the same output from a set of ordered batches.

Some systems include "housekeeping" threads that asynchronously modify application server state. For example, an HDFS server maintains a list of live data servers, removing an uncommunicative server from the list after a timeout. An application must ensure that housekeeping threads run at well-defined points in the sequence of requests by, for example, scheduling such threads at specific points in virtual time rather than at periodic real time intervals.

7.2 Checkpoint Generation

In an asynchronous system, even correct server replicas can fall arbitrarily behind, so state machine replication frameworks must provide a way to checkpoint a server replica's state, to certify that a quorum of server replicas have produced identical checkpoints, and to transfer a certified checkpoint to a node that has fallen behind [18]. Recall from the discussions in Chapters 4 and 6 that the UpRight library periodically tells the server application to checkpoint its state to persistent memory and asks for a cryptographic hash that uniquely identifies that stable state. Further, if a replica falls behind, the library (i.e., the server shim at that replica) communicates with the other server shims to retrieve the most recent checkpoint, restarts the server application using that state, and finally replays the log of ordered requests

after that checkpoint to bring the replica to the current state.

Given this context for how and when application checkpoints are generated and applied, there are several pragmatic concerns to consider. Application checkpoints must be (1) inexpensive to generate because the replication framework requests checkpoints at a high frequency, (2) inexpensive to apply because the replication framework uses checkpoints in both the rare case of a machine crashing and restarting and the more common case of a machine falling behind on message processing, (3) deterministic because correct nodes must generate identical checkpoints for a given request sequence number, and (4) nonintrusive on the codebase because we must not require extensive modifications of applications.

There is tension among these requirements. For example, generating checkpoints more frequently increases generation cost but reduces recovery time (because the log that must be applied will be correspondingly shorter.) For example, requiring an application to store its data structures in a memory array checksummed with a Merkle tree [18] can reduce checkpoint generation and fetch time (since only changed parts need be stored or fetched) but may require intrusive changes to legacy applications.

We resolve this tension through a generic checkpoint *glue library* that implement a checkpoint/delta approach and relies on a helper process for deterministic checkpoint generation. The checkpoint/delta approach allows the generic checkpoint glue to provide the UpRight library with the required frequent checkpoints while only rarely paying the high cost of generate the native application checkpoints.

We use the generic checkpoint glue with both HDFS and ZooKeeper. The generic checkpoint glue is suitable for use with other applications, though an application specific glue can implement a different checkpoint strategy [18, 104] if needed.

Checkpoint/delta approach. The checkpoint/delta approach seeks to minimize intrusiveness to legacy code by reusing existing application functionality and interposing a small amount of batch logging.

We posit that most crash fault tolerant services will already have some means to checkpoint their state. So, to minimize intrusiveness, to lower barriers to adoption, and to avoid the need for projects to maintain two distinct checkpoint mechanisms, we wish to use applications' existing checkpoint mechanisms. Unfortunately, the existing application code for generating checkpoints is likely to be suitable for

infrequent, coarse grained checkpoints. For example, both the HDFS and Zookeeper applications produce their checkpoints by walking their important in-memory data structures and writing their contents to persistent memory.

The checkpoint/delta approach uses existing application code to take checkpoints at the approximately the same coarse-grained intervals the original systems use. We presume that these intervals are sufficiently long that the overhead is acceptable. To produce the more frequent checkpoints required by the UpRight shim, the glue library augments these infrequent, coarse-grained, application checkpoints with frequent fine-grained deltas. A delta is the log of batches since the previous delta; the log of deltas compose to form a log of batches from one checkpoint to the next. Figure 7.2 presents the checkpoint/delta approach graphically.

A naive implementation of the checkpoint/delta approach produces checkpoints as shown in Figure 7.2. Specifically, each time a coarse-grained checkpoint is produced, that checkpoint is returned to the library. This naive approach has two fundamental limitations. First, it can introduce periodic latency spikes into the system if generating the coarse-grained checkpoint is a very expensive operation. Second, if an execution replica begins loading a checkpoint/delta around the same time that a coarse-grained checkpoint is produced, the replica is likely to fetch both the new and old coarse-grained checkpoints.

We avoid these two issues by structuring the checkpoints produced by the checkpoint/delta approach in a similar fashion to the checkpoint and batch logs maintained by the order stage (Section 5.3.1): a checkpoint consists of one course-grained checkpoint and sufficient deltas to reach the next course-grained checkpoint, but not enough to reach the subsequent course-grained checkpoint, as shown in Figure 7.3. This increases the time budget available to the application to produce the coarse-grained checkpoint before it is needed by the system. It also makes it possible for a recovering replica to load a single coarse-grained checkpoint and as many logs as necessary to catch up with the rest of the system, even if multiple coarse-grained checkpoints are generated while the recovery takes place.

Within the checkpoint/delta approach, the application's checkpoints must be produced deterministically. We overview several approaches below: helper processes, stop and copy, OS fork, and application copy-on-write. We use the helper process approach in our HDFS and ZooKeeper prototypes.

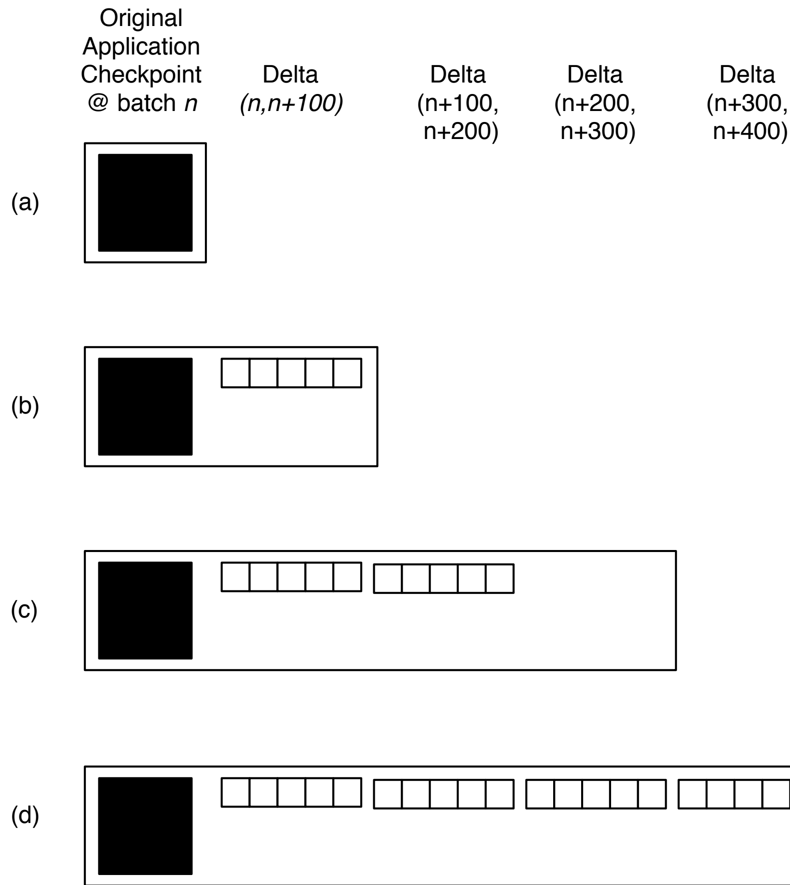

Figure 7.2: The checkpoint/delta approach for managing application checkpoints. Original application checkpoints are taken infrequently, but the library requests a checkpoint every 100 batches. (a) shows the original application checkpoint taken after executing batch n. (b) shows the checkpoint returned to the replication library after executing batch $n+100$. This checkpoint consists of the application checkpoint at n and the log of the next 100 batches. (c) shows the checkpoint returned to the replication library after executing batch $n+200$. (d) shows the checkpoint returned to the replication library after executing batch $n+400$.

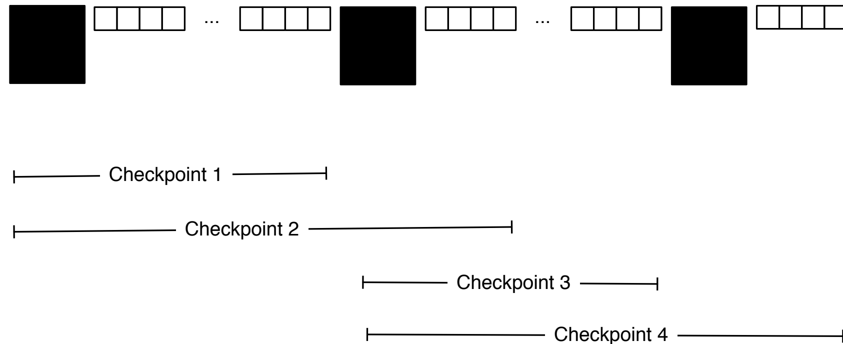

Figure 7.3: Checkpoint-deltas returned to the application. Each returned checkpoint-delta consists of a coarse grained application checkpoint and sufficient deltas to produce the next coarse grained checkpoint.

Helper process. The helper process approach produces checkpoints asynchronously to avoid pausing request execution and seeks to minimize intrusiveness to legacy code.

To ensure that different replicas produce identical checkpoints without having to pause request processing, each node runs two slightly modified instances of the server application process—a primary and a helper—to which we feed the same series of requests. We deactivate the checkpoint generation code at the primary. For the helper, we omit sending replies to clients, and we pause the sequence of incoming requests so that it is quiescent while it is producing a checkpoint.

The helper process approach requires us to run two copies of the application at each replica. Surprisingly, our experiences with ZooKeeper and HDFS indicate that the overheads of this approach are not unmanageable.

Stop and copy. If an application's state is small and an application can tolerate a few tens of milliseconds of added latency, the simplest checkpoint strategy is to pause the arrival of new requests so that the application is quiescent while it writes it state to disk. Since we eliminate other sources of nondeterminism as described above, this approach suffices to ensure that replicas produce identical checkpoints for a given sequence number.

Unfortunately, stop and copy is not suitable for applications that either (a) have a large amount of application state or (b) are not compatible with periodic

latency spikes.

OS fork. Operating systems provide a *fork()* call that can be used to make an instantaneous copy of a process. One approach is to use *fork()* to create a copy of the application and then generate the checkpoint from the copy before destroying the auxiliary process.

Unfortunately, on most operation systems *fork()* does not interact properly with the JVM and it is not uncommon to see the child process crash due to unfortunately timed garbage collection or some other background process.

Application copy on write. Rather than use a helper process to produce a deterministic checkpoint, applications can be modified so that their key data structures are treated as copy on write while checkpoints are taken [19, 18, 86]. This approach can have lower performance overheads, but can require extensive application modification to support.

7.3 HDFS case study

The Hadoop Distributed File System (HDFS) [43] is an open-source cluster file system modeled loosely on the Google File System [39]. It provides parallel, high-throughput access to large, write-once, read-mostly files.

An HDFS deployment comprises a single *NameNode* and many *DataNodes*. Files are broken into large (default 64MB) blocks, and by default each block is stored on three DataNodes. The NameNode keeps the file name to block ID mappings and caches the block ID to DataNodes mappings reported by DataNodes as soft state. We overview the interactions between NameNodes, DataNodes, and clients in Section 7.3.1.

UpRight-HDFS enhances HDFS by (1) eliminating a single point of failure and improving availability by supporting redundant NameNodes with automatic failover and (2) providing end-to-end Byzantine fault tolerance against faulty clients, DataNodes, and NameNodes.

7.3.1 Baseline system

In this section we overview the basic operation of HDFS.

To write a new block, a client requests a new block ID from the NameNode, the NameNode selects a block ID and a list of DataNodes, the client sends a write comprising the block ID, the data, a list of 4-byte CRC32 checksums for each 512 bytes of data, and a list of DataNodes to the nearest listed DataNode, that DataNode stores the data and checksums, forwards the write to the next DataNode on the list, and reports the completed write to the NameNode. After the DataNodes acknowledge the write, the client sends a *write complete* request to the NameNode; the write complete request returns once the NameNode knows that the data has reached the required number of DataNodes. To read a block, a client requests a list of the block's DataNodes from the NameNode, sends the read request to a DataNode, and gets the data and checksums in reply.

DataNodes send periodic heartbeats to the NameNode. After a number of missed heartbeats, the NameNode declares the DataNode dead and replicates the failed node's blocks from the remaining copies to other DataNodes.

The NameNode checkpoints its state to a file with the help of a Secondary NameNode. The NameNode writes all transactions to a series of log files. Periodically, the Secondary fetches the most recent log file and the current checkpoint file. The Secondary then loads the checkpoint, replays the log, and writes a new checkpoint file. Finally, the Secondary sends the new checkpoint file back to the NameNode, and the NameNode can reclaim the corresponding log file. If a NameNode crashes and recovers, it first loads the checkpoint and then replays the log.

The fault tolerance of the baseline HDFS system is not cleanly categorizable as "crash" or "Byzantine." The checksums at the DataNodes protect against some but not all Byzantine failures. For example, if a DataNode suffers a fault that corrupts a disk block but not the corresponding checksum, then a client would detect the error and reject the data, but if a faulty DataNode returns the wrong block and also returns the checksum for that wrong block, a client would accept the wrong result as correct. In its default configuration, HDFS can ensure access to all data even if two DataNodes fail by omission, and it can ensure that it returns correct data for some but not all commission failures of up to two DataNodes. We will summarize HDFS DataNodes' fault tolerance as $u = 2$ $r = 0/2$.

HDFS's Secondary NameNode's role is just to compact the log into the checkpoint file, and there is no provision for automatically transferring control from the NameNode to the Secondary NameNode. If the NameNode suffers a catastrophic

failure, one could imagine manually reconfiguring the system to run the NameNode on what had been the Secondary's hardware, but recent updates could be lost. An HDFS NameNode's fault tolerance is $u = 0$ $r = 0$.

7.3.2 UpRight-HDFS

Given the UpRight framework, adding Byzantine fault tolerance to HDFS is straightforward.

UpRight-NameNode

Adapting the HDFS NameNode to work with UpRight requires modifications to less than 1750 lines of code. The bulk of these changes, almost 1600 lines, relates to checkpoint management and generation. In particular, we add about 730 lines to include additional state in checkpoints. For example, we include mappings from block IDs to DataNodes in a NameNode's checkpoints—although we still treat these mappings as soft state that expires when a DataNode is silent for too long, including this state in the checkpoint ensures that NameNode replicas processing a request agree on whether the state has expired or not. In addition, we add about 830 lines to modify the logs to record every operation that modifies any NameNode state rather than only the modifications to the file ID to block ID mapping.

The other major change needed to make the HDFS NameNode compatible with UpRight is removing sources of nondeterminism from its request execution path. These changes affect under 150 lines and fall into 3 categories. We replace 5 references to local system time with references to the time provided by the order nodes for the current batch of request. Similarly, we modify 20 calls to *random()* so that they are all seeded by the agreed upon order time. The final step to removing nondeterminism is disabling the threads responsible for running a variety of periodic background jobs based on *System.time()* and instead executing those tasks based on the time specified by the order nodes.

Clients. The modified HDFS NameNode corresponds to the *application server* in the UpRight library deployment. When deploying the service, we treat both HDFS clients and and HDFS DataNodes as *application clients*. Reads and writes issued by HDFS clients are processed as client requests in the UpRight library. Similarly,

DataNode heartbeats and notifications that a write has completed are processed as client requests in the UpRight library.

UpRight-DataNode

We originally imagined that we would replicate each DataNode as a BFT state machine and reduce the application-level data replication in light of the redundancy in the BFT DataNode "supernodes." Although academically pure, simply using a black box state machine replication library to construct BFT data nodes would have changed the replication policies of the system in significant and perhaps undesirable ways. For example, HDFS's default data placement policy is to store the first copy on a node in the same rack as the writer, the second copy on a node in another rack, the third copy on a different node in the same rack as the second, and additional copies on randomly selected, distinct nodes. Further, if a DataNode fails and is replaced, HDFS ends up spreading the recovery cost approximately evenly across the remaining DataNodes. Additionally, if a new DataNode is added, the system gradually makes use of it. Although one could imagine approximating some of these policies within a state machine replication approach, we instead leave the (presumably) carefully-considered HDFS DataNode replication policies in place (i.e., 3-way replication). These policies ensure that block writes complete if at most $u = 0$ of the selected DataNodes are faulty and reads complete if at most $u = 2$ of the selected DataNodes are faulty. Our modifications further ensure that a reads only return correct values, i.e., $r = 3$.

To that end, our UpRight-DataNode makes a few simple changes to the existing DataNode. The main changes are to (1) add a cryptographic subblock hash on each 64KB subblock of each 64MB (by default) block and a cryptographic block hash across all of a block's subblock hashes and (2) store each block hash at the NameNode. In particular, DataNodes compute and store subblock and block hashes on the writes they receive, and they report these block hashes to the NameNode when they complete the writes. A client includes the block hash in its write complete request to the NameNode, and the NameNode commits a write only if the client and a sufficient number of DataNodes report the same block hash. As in the existing code, clients retry on timeout, the NameNode eventually aborts writes that fail to complete, and the NameNode eventually garbage collects DataNode blocks that are

not included in a committed write.

To read a block, a client fetches the block hash and list of DataNodes from the NameNode, fetches the subblock hashes from a DataNode, checks the subblock hashes against the block hash, fetches subblocks from a DataNode, and finally checks the subblocks against the subblock hashes; the client retries using a different DataNode if there is an error.

These changes require us to change or add 189 LOC at the client, 519 lines at the DataNode, and 238 lines at the NameNode.

Finally, we add the expected MACs and MAC authenticators to all messages with the exception of subblock hash and subblock data read replies from DataNodes to clients, which are directly or indirectly checked against the block hash from the NameNode.

Programmer background. The modifications to HDFS were performed by a junior graduate student (Sangmin Lee) with minimal knowledge of the internals of the UpRight library. Development took a total of approximately three months, most of that time was spent learning how the internals of the HDFS codebase work.

7.3.3 Evaluation

In this section we compare UpRight-HDFS with the original. Unless otherwise noted, experiments run on subsets of 107 Amazon EC2 *small* instances [6]. In each experiment, we have 50 DataNodes and 50 clients, and each client reads or writes a series of 1GB files. For both systems, we replicate each block to 3 DataNodes, giving $u = 2$, $r = 2/0$ for HDFS and $u = 2$ $r = 2$ for UpRight. HDFS's NameNode is a single point of failure ($u = r = 0$). For the UpRight-HDFS runs, we configure the NameNodes for $u = r = 1$ and co-locate the RQ and order nodes. To evaluate UpRight's ability to support CFT configurations, we also look at a $u = 1$ $r = 0$ configuration.

Figure 7.4 shows the throughput achieved with 50 clients and DataNodes. For both systems, write throughput is lower than read throughput because each block is written to three disks but read from one. Even with $r = 1$, UpRight-HDFS's read performance is approximately equal to that of HDFS's because only one DataNode is required to read and send the data. With $r = 1$, UpRight-HDFS's write performance is over 70% of HDFS's; the slowdown on writes appears to be

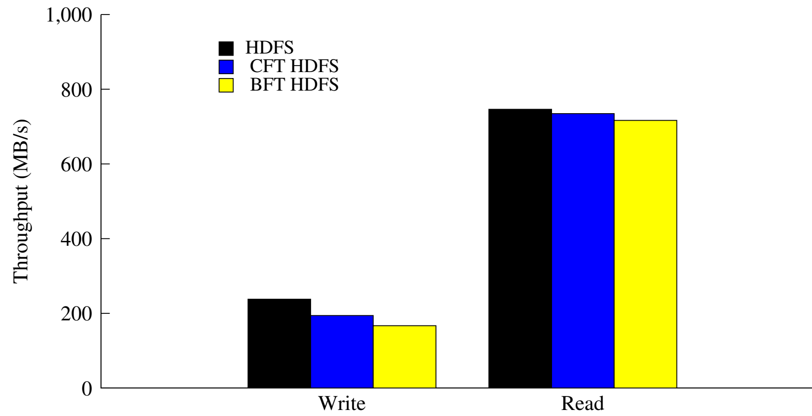

Figure 7.4: Throughput for HDFS and UpRight-HDFS.

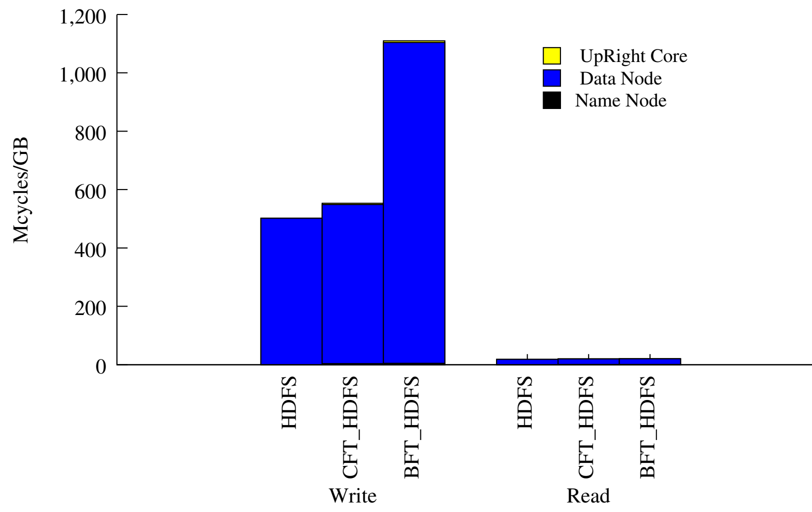

Figure 7.5: CPU consumption (jiffies per GB of data read or written) for HDFS and UpRight-HDFS.

due to added agreement for the replicated NameNode and the overheads of MAC computations for the DataNodes. With $r = 0$, the MAC computations are omitted and write performance is over 80% of HDFS's; the compensation for this slowdown is the ability to remain available even if a NameNode crashes.

Figure 7.5 shows the CPU consumption for these workloads. When $r = 1$, UpRight-HDFS's CPU costs are within a factor of 2.5 of the original for writes and within a factor of two for reads. Note that CPU consumption is one of the worst metrics for UpRight-HDFS; other system resources like the disks and networks have much lower overheads. When $r = 0$, the overheads are smaller—factors of 1.1 and 1.6 for writes and reads, respectively. We also note that the computational cycles for these workloads are dominated by the work performed at the DataNodes and not the NameNode replicated with the UpRight library.

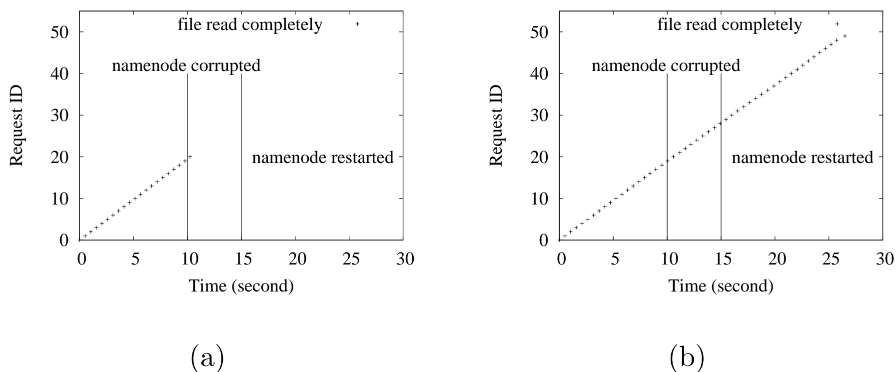

Figure 7.6: Completion time for requests issued by a single client. In (a), the HDFS NameNode fails and is unable to recover. In (b), a single UpRight-HDFS NameNode fails, and the system continues correctly.

UpRight-HDFS incurs additional computational overheads for lower performance than HDFS. These costs come with a benefit as demonstrated by Figure 7.6. The two graphs plot completion time for requests issued by a single client that issues each request .5 seconds after the previous request completes. After 10 seconds of this workload we kill a NameNode and in the process corrupt its checkpoint log. We then restart the NameNode after an additional 5 seconds. Progress with the HDFS NameNode stops at 10 seconds when the log becomes corrupted. When the NameNode restarts 5 seconds later it immediately crashes again after attempting

to load the corrupted log. In UpRight-HDFS, the absence of a single NameNode does not prevent progress. Additionally, when the failed NameNode restarts, it fetches a valid state from the other replicas and resumes correct operation rather than attempting to load its corrupted local log.

7.3.4 MapReduce

MapReduce is an application frequently run on top of HDFS. In Figure 7.7 we report the execution times of the TeraSort and TeraGen MapReduce workloads. TeraGen generates 100,000,000 random 100 byte entries and TeraSort sorts the generated data.

This set of experiments is run on a collection of 4 core 2.4Ghz processors and 8GB of RAM. There are 20 DataNodes in the experiments, with 20 map tasks for TeraGen and 20 reducers for TeraSort all running on the DataNodes. HDFS is configured to use 3 way data replication.

Our current UpRight implementation allows clients to have at most one request outstanding at any time and uses a single proxy client per machine, regardless of how many tasks are running on that machine. In this experiment, each mapper, reducer, and DataNode process on a single machine shares one UpRight client proxy.

Our results indicate that the UpRight library imposes a modest overhead on overal execution time. We believe this overhead can be reduced by improving the implementation of both the UpRight library and the interactions between the application and library at the client side application. Specifically, we believe that engineering UpRight to support multiple outstanding requests per client or to have a client per task rather than a single client per machine would improve performance.

7.4 ZooKeeper case study

ZooKeeper [108] is an open-source coordination service that, in the spirit of Chubby [12], provides services like consensus, group management, leader election, presence protocols, and consistent storage for small files.

ZooKeeper guards against omission failures. However, because data centers typically run a single instance of a coordination service on which many cluster services depend [19], and because even a small control error can have dramatic

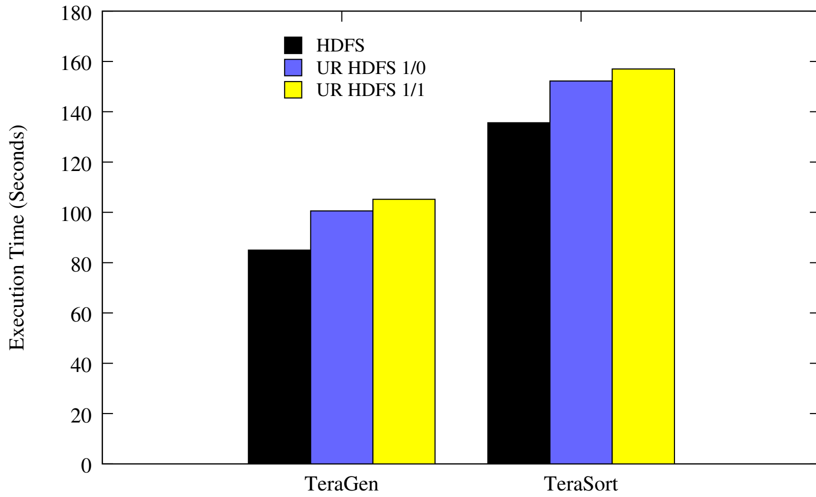

Figure 7.7: Execution time for TeraGen and TeraSort MapReduce workloads.

effects [97], investing modest additional resources to protect the service against a wider range of faults may be attractive.

7.4.1 Baseline system

A ZooKeeper deployment comprises $2u + 1$ servers; a common configuration is 5 servers for $u = 2$ $r = 0$. Servers maintain a set of hierarchically named objects in memory. Writes are serialized via a Paxos-like protocol, and reads are optimized to avoid consensus where possible [18]. A client can set a *watch* on an object so that it is notified if the object changes unless the connection from the client to a server breaks, in which case the client is notified that the connection broke.

For crash tolerance, each server synchronously logs updates to stable storage. Servers periodically produce *fuzzy snapshots* to checkpoint their state: a thread walks the server's data structures and writes them to disk, but requests concurrent with snapshot production may alter these data structures as the snapshot is produced. If a ZooKeeper server starts producing a snapshot after request s_{start} and finishes producing it after request s_{end}, the fuzzy snapshot representing the system's state after request s_{end} comprises the data structures written to disk plus the log of updates from s_{start} to s_{end}.

7.4.2 UpRight-ZooKeeper

UpRight-ZooKeeper is based on ZooKeeper version 3.0.1. Given the UpRight framework, adding Byzantine fault tolerance to ZooKeeper to produce UpRight-ZooKeeper is straightforward. Our shims use standard techniques to add authenticators to messages and to send/receive them to/from the right quorums of nodes. We use the techniques described above to support watches via server push, to make time-based events happen deterministically across replicas at the same virtual time, and to canonicalize read-only replies. ZooKeeper's fuzzy snapshots align well with our hybrid checkpoint/delta approach; we modify ZooKeeper to make the snapshots deterministic and identical across replicas using helper-process approach.

The original ZooKeeper server comprises 13589 lines of code (LOC). We add or modify 604 lines to integrate it with UpRight. The bulk of these changes involved modifying the checkpoint generation code to include all required state and integrate a helper process for use with the hybrid checkpoint/delta approach (347 LOC), glue code to handle communication between ZooKeeper and the UpRight and checkpoint/delta libraries (129 LOC), and making references to time and randomness deterministic across replicas (66 LOC). We also deactivate or delete some existing code. In particular, we delete 342 LOC that deal with asynchronous IO and multithreading, and we no longer use 5644 LOC that handle ZooKeeper's original replication protocols.

Programmer background. The modifications to ZooKeeper were performed by a pair of junior graduate students (Manos Kapritos and Yang Wang) with minimal knowledge of the internals of the UpRight library. Development took a total of approximately three months, most of that time was spent learning how the internals of the ZooKeeper codebase work.

7.4.3 Evaluation

We evaluate ZooKeeper 3.0.1 and UpRight-ZooKeeper running on the hardware described in Section 6.5. For ZooKeeper, we run with the default 5 servers ($u = 2$ $r = 0$). We then configure UpRight-ZooKeeper to tolerate as many or more faults. In particular, we examine UpRight-ZooKeeper with $u = 2$ $r = 1$ for all phases to minimize the replication cost of adding commission failure tolerance while

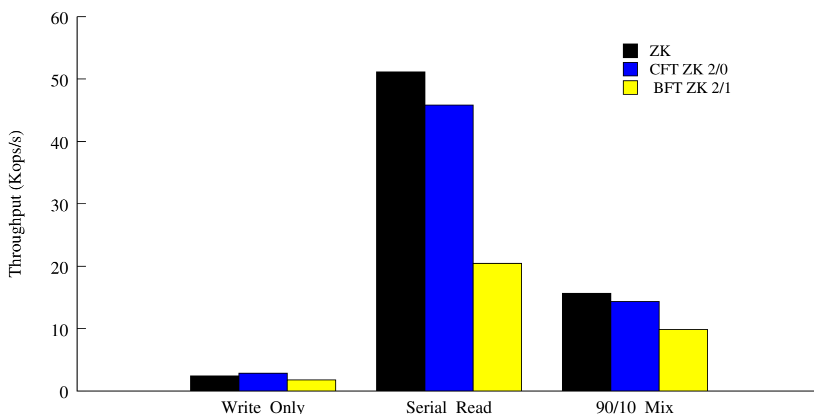

Figure 7.8: Throughput for UpRight-ZooKeeper and ZooKeeper for workloads comprising different mixes of 1KB reads and writes.

retaining at least ZooKeeper's original omission failure tolerance. We also examine a configuration that we refer to as u =2+ $r = 1$ that has $u = 2$ $r = 1$ for the RQ and order stages and $u_{exec} = 3$ $r_{exec} = 1$ for the execution stage; this configuration retains ZooKeeper's default 5 execution replicas. The results presented here rely on the helper process approach for checkpointing. We observe similar performance when using copy on write techniques.

In addition, we evaluate UpRight-ZooKeeper's performance in CFT configurations ($r = 0$) to explore whether UpRight would be a suitable for new applications that want to support both CFT and BFT configurations using a single library. We evaluate the performance of UpRight-ZooKeeper with $u = 2$ $r = 0$ to match ZooKeeper's omission tolerance with the minimum degree of replication. We also evaluate a configuration that we refer to as u =2+ $r = 0$ that has $u = 2$ $r = 0$ for the RQ and order stages and $u_{exec} = 4$ $r_{exec} = 0$ for the execution stage; this configuration retains ZooKeeper's default 5 execution replicas.

Figure 7.8 shows throughput for different mixes of 1KB reads and writes.

For writes, the systems sustain several thousand requests per second. Nearly a decade of effort to improve various aspects of BFT agreement [1, 18, 24, 26, 49, 50, 92, 100, 104, 107] have paid off: when $r = 1$, UpRight-ZooKeeper's write throughput is 77% of ZooKeeper's for both $u = 2$ and u =2+. UpRight also appears to provide competitive write performance for CFT configurations: for $u = 2$ or u =2+ and $r = 0$ UpRight-ZooKeeper's throughput with $r = 0$ and either $u = 2$ or u =2+ is

more than 111% of ZooKeeper's.

For reads that can accept serializability for their consistency semantics, both ZooKeeper and UpRight-ZooKeeper exploit the read-only optimization to skip agreement and issue requests to a quorum of $r + 1$ execution nodes that have processed the reader's most recent write. Both systems' read throughputs are many times their write throughputs, but in configurations where ZooKeeper queries fewer execution nodes or has more total execution nodes, its peak throughput can be proportionally higher. For example, when ZooKeeper sends read requests to 1 server and spreads these requests across 5 execution replicas, we expect to see about 2.5 times the throughput of a configuration where UpRight-ZooKeeper sends read requests to 2 servers (for $r = 1$) and spreads them across 4 execution replicas. When UpRight-ZooKeeper is configured to tolerate commission failures, it pays additional CPU overheads for cryptographic checksums but saves some network overheads by having only one execution node send a full response and having the others send a hash [18]. Overall, UpRight-ZooKeeper's serializable read throughput ranges from 17.5 Kops/s to 43.4 Kops/s, which is 34% to 85% of ZooKeeper's 51.1 Kops/s throughput.

Although reading identical results from a properly chosen quorum of $r + 1$ servers can guarantee that the read can be sequenced in a global total order, the position in the sequence may not be consistent with real time: a read by one client may not reflect the most recently completed write by another. So, some applications may opt for the stronger semantics of linearizability. For linearizable reads, UpRight-ZooKeeper can still use the read only optimization, but it must increase the read quorum size to $n_{exec} - r_{exec}$. To enforce linearizability the original ZooKeeper issues a *sync* request through the agreement protocol and then issues a read to the same server, which ensures that server has seen all updates that completed before the sync.

The last group of bars examines performance for a mix of 90% serializable reads and 10% writes. When UpRight-ZooKeeper is configured to tolerate $r = 1$ commission failures, its performance is over 66% of ZooKeeper's. When it is configured to tolerate omission failures only, its performance is comparable to ZooKeeper's.

Although the throughputs of our BFT configurations are comparable to those of the original CFT system, the extra guarantees come at a cost of resource consumption. Figure 7.9 shows that each request consumes significantly more CPU cycles

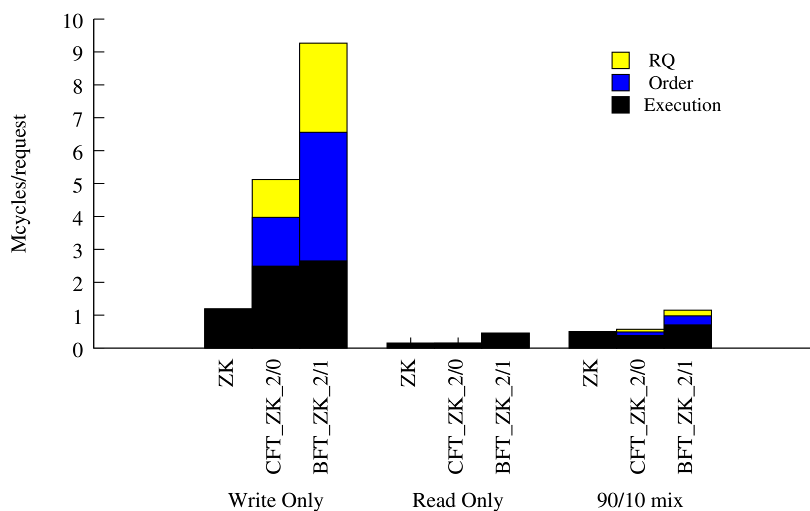

Figure 7.9: Per-request CPU consumption for UpRight-ZooKeeper and ZooKeeper for a write-only workload. The y axis is in jiffies. In our system, one jiffy is 4 ms of CPU consumption.

under UpRight-ZooKeeper than under ZooKeeper. The graph shows per-request CPU consumption when both systems are heavily loaded; we observe similar results across a wide range of loads.

We note that although using Java rather than C for agreement only modestly hurts our throughput for this application, it does significantly increase our resource consumption. Judging by peak throughputs on similar hardware, agreement protocols like PBFT and Zyzzyva may consume an order of magnitude fewer CPU cycles per request than our Zyzzyvark implementation. Future work is needed to see if a C realization of UpRight's agreement protocol would provide a lower cost option for deployments willing to shift from Java to C.

Figure 7.10 shows how throughput varies over time as nodes crash and recover. For this experiment we compare against ZooKeeper 3.1.1 because it fixes a bug in version 3.0.1's log garbage collection that prevents this experiment from completing. The workload is a series of 1KB writes generated by 16 clients, and we compare ZooKeeper ($u = 2$ $r = 0$) with UpRight-ZooKeeper configured with u =2+ $r = 1$. At times 30, 270, 510, 750, and 990 we kill a single execution node and restart it 60 seconds later. At time 1230 we kill all execution nodes and restart them 20 seconds later. Both systems successfully mask partial failures and recover

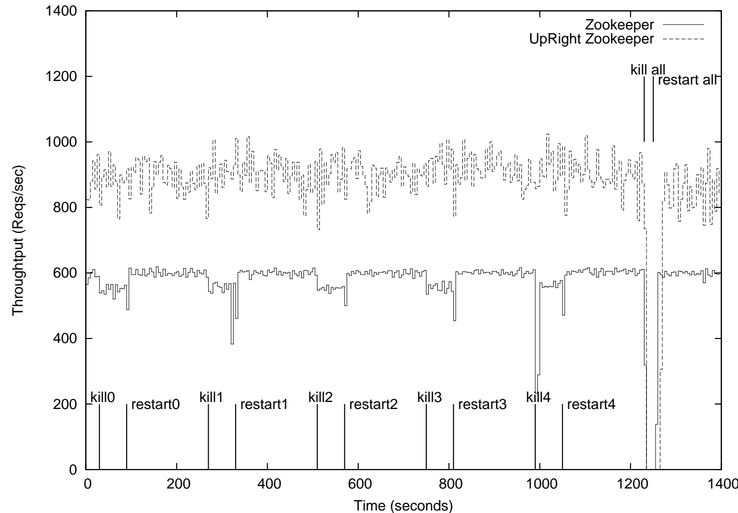

Figure 7.10: Performance v. time as machines crash and recover for ZooKeeper and UpRight-ZooKeeper.

quickly after a system-wide crash-recover event.

7.5 Conclusion and Discussion

In this chapter we relate our experience modifying HDFS and ZooKeeper to be compatible with the UpRight library. We take three lessons from our experience.

First, the changes required to make an existing (Java) application UpRight compliant are modest in scope and do not require extensive knowledge of (BFT) replication. In concrete terms, we modified approximately 2500 lines of code (out of 37,000) in HDFS (between the NameNode and the DataNode) and 600 lines of code (out of 13,500) in ZooKeeper. These modifications were made by junior students that did not know the details of the replication library. They reported that the ability to make the system fail stop by setting $u = 0$ and $r = 1$ facilitated their development by highlighting the presence of non-determinism and aiding in identifying the source of the non-determinism. We believe this is a step forward in comparison to previous replication libraries that are either integrated tightly into the

application (i.e., ZooKeeper [108], Q/U [1], and Chubby [12]) or require extensive application modifications to fit a library defined memory model (i.e., PBFT [18, 86], Zyzzyva [49], Aardvark [24], and others [50, 104, 107]).

Second, building a replication library to provide UpRight fault tolerance transforms the question of Byzantine or crash fault tolerance from a design decision to a configuration decision. With a single library, and a single application code base, we are able to provide Byzantine, crash, or hybrid fault tolerance. We believe this is an important step to facilitating adoption of BFT replication in production applications.

Finally, the performance of UpRight applications can be competitive with the original code bases. Despite the fact that we made the conscious decision to keep the application modifications simple and to reuse application functionality when possible rather than optimizing the application and environment, we observe that the performance of an UpRight system is within 25% of the performance of the original system for most workloads.

Chapter 8

Background and state machine replication

There is a large body of research on fault tolerance and state machine replication. This thesis builds on much of that work and refines and incorporates ideas developed by a multitude of other researchers.

Section 8.1 discusses the foundations of state machine replication. Section 8.2 discusses a variety of work on consensus and quorum systems that lies at the core of most RSM protocols. Section 8.3 discusses contemporary replication libraries developed as part of the effort to demonstrate that Byzantine fault tolerance and poor performance are not synonymous. Section 8.4 discusses previous work related to the performance of fault tolerant systems in the presence of failures. Section 8.5 discusses current commercial best practices for building reliable systems.

8.1 RSM approach

State machine replication is a powerful technique for building reliable services from faulty components [52, 88]. The basic idea behind state machine replication is simple: as long as every replica executes the same sequence of requests then correct replicas will provide the same responses to those requests and the collection of potentially faulty components can be viewed as a single correct node. There is a large body of previous work on the development of asynchronous replicated state machine (RSM) prototypes [1, 18, 24, 26, 49, 50, 92, 100, 104, 107] and deployed

systems [12, 108] based on the Paxos RSM protocol [53].

The primary objective of RSM protocols is to ensure that the end-to-end service remains both safe, e.g. correct, and live, e.g. available, despite the failure of individual replicas. The network connecting replicas in these systems is assumed to be asynchronous and consequently capable of arbitrarily delaying, reordering, or dropping messages. In asynchronous environments where nodes are allowed to fail, it is impossible to insure that non-trivial systems will remain both safe and live [35]. RSM protocols are consequently designed to be *fault tolerant*. A protocol is fault tolerant if, despite a bounded number of failures, it is (1) safe always and (2) live provided that the network is sufficiently well behaved.

8.2 Consensus

The core unit of every RSM protocol is a consensus, or agreement, protocol. There is a large body of work on synchronous [79, 61, 27, 84, 64, 38, 47, 29, 59] and asynchronous [11, 15, 33, 65, 30, 13, 54, 56, 53, 54, 57, 58, 60, 69, 70, 68, 32, 35] consensus that establishes when it is possible to solve consensus and the number of replicas required.

While the full body of previous work on consensus informs our design and implementation, the work by Lamport [56, 60], Dutta et al. [32], and Martin et al. [68] is especially important. These works explore circumstances when the standard $3f + 1$ replicas are not required to solve consensus and provide the foundation for the protocols we use to replicate the authentication, order, and execution stages of the UpRight architecture.

8.3 Recent RSM history

Our work builds on a number of previous asynchronous RSM prototypes [18, 42, 86, 1, 26, 45, 50, 49, 92, 104, 107].

Historically, BFT state machine replication was widely considered to be inefficient and fundamentally inappropriate for use in deployed systems. This belief held until Castro and Liskov provided a practical BFT NFS implementation [18]. Their protocol, PBFT, is based on a three-phase commit protocol that uses MACs, rather than digital signatures, for message authentications. Many subsequent BFT

RSM systems [86, 26, 42, 50, 49, 92, 104, 107] are inspired by PBFT.

The systems directly inspired by PBFT can be broken down into three categories. Systems in the first set [26, 49, 92, 42] attempts to optimize throughput and latency by taking advantage of situations in which Byzantine consensus can be solved using two, rather than three, phase commit [42, 49, 92] or without requiring any all-to-all communication steps [26, 42]. Systems in the second set [104, 107] reduce deployment costs by leveraging the disparity between the number of replicas required to agree on the order of requests and the number of replicas required to execute the requests. Systems in the third set optimize performance by facilitating parallel execution [50] or simplify development [86] through an object based API. The UpRight architecture extends the separation of agreement and execution emplyed by the second set of systems while the replicated order stage is based on similar techniques to those developed in the first set of systems. The work in the third set of systems is orthogonal to the UpRight library.

Another thread of previous work [45, 1] differentiates itself from the PBFT lineage by explicitly basing its replication protocols on quorums, rather than consensus. While the protocols at the core of these systems do not share many obvious similarities with the systems in the PBFT lineage, careful consideration of generalized Paxos consensus [53] and the various special case replication requirements for implementing consensus in two step [32, 56, 68] indicate that the underlying quorum protocols are in fact very specific special cases of consensus. The protocols we use for the replicated authentication and execution stages are similar to the quorum protocols used in this lineage of work.

8.4 Performance with failures

We are not the first to notice significantly reduced performance for BFT protocols during periods of failures or bad network performance or to explore how timing and failure assumptions impact performance and liveness of fault tolerant systems.

Singh et al. [95] show that PBFT [18], Q/U [1], HQ [26], and Zyzzyva [49] are all sensitive to network performance. They provide a thorough examination of the gracious executions of the four canonical systems through a ns2 [76] network simulator. Singh et al. explore performance properties when the participants are well behaved and the network is faulty; we focus our attention on the dual scenario

where the participants are faulty and the network is well behaved.

Aiyer et al. [4] and Amir et al. [7] note that a slow primary can result in dramatically reduced throughput. Aiyer et al. combat this problem by frequently rotating the primary. Amir et al. address the challenge instead by introducing a pre-agreement protocol requiring several all-to-all message exchanges and using signatures for all authentication. Condie et al. [25] address the ability of a well placed adversary to disrupt the performance of an overlay network by frequently restructuring the overlay, effectively changing its view.

The signature processing and scheduling of replica messages in Aardvark is similar in flavor to the early rejection techniques employed by the LOCKSS system [40, 66] in order to improve performance and limit the damage an adversary can inflict on system.

8.5 Application fault tolerance

Commercial best practices for replication have evolved towards increasing tolerance to fail-stop faults as hardware costs fall, as replication techniques become better understood and easier to adopt, and as systems become larger, more complex, and more important. For example, once it was typical for storage systems to recover from media failures using off-line backups; then single-parity or mirrored RAID [20] became *de rigeur*; now, there appears to be increasingly routine use of doubly-redundant storage [39, 90, 81]. Similarly, although two-phase commit is often good enough—in the absence of commission failures it can be always safe and rarely unlive—increasing numbers of deployments pay the extra cost to use Paxos [53, 77] three-phase commit [12, 99] to simplify their design or avoid corner cases requiring operator intervention [12].

Failed processes and hardware are not always polite enough to stop cleanly. Instead, they may continue to operate and provide incorrect outputs or corrupt internal state for a variety of reasons including bad NICs [2], soft CPU errors [94], memory errors [19], disk errors [82, 90, 81], and software Heisenbugs [105].

Deployed systems increasingly include limited Byzantine fault tolerance aimed at high-risk subsystems. For example the ZFS [85], GFS [39], and HDFS [43] file systems provide checksums for on-disk data [82]. As another example, after Amazon S3 was felled for several hours by a flipped bit, additional checksums on system

state messages were added [97]. Although it may be cheaper to check for and correct faults at critical points than to do so end-to-end, we fear that it may be difficult to identify all significant vulnerabilities *a priori* and complex to solve them case by case with *ad hoc* techniques. We demonstrate that end-to-end techniques can be applied to existing applications.

Chapter 9

Conclusion

This thesis describes the design, implementation, and deployment of the UpRight replication library. More importantly, it presents a concrete step towards making Byzantine fault tolerance a deployable option for general computing systems.

We believe that the UpRight library eases the path to adopting Byzantine fault tolerance in two important ways. First, the UpRight library provides both crash and Byzantine faul tolerance in a single code base. Flexible fault tolerance encourages incremental adoption of Byzantine fault tolerance by removing the need to maintain multiple code bases and allowing sysadmins to "add" Byzantine fault tolerance to an existing system by adding additional resources and changing a configuration parameter rather than deploying and supporting an entirely different system. Second, the application interface provided by the UpRight library is not onerous; our experience indicates that programmers unfamiliar with the details of the replication library can port legacy applications with only nominal effort.

In addition to the practical benefits mentioned above, this thesis makes three important conceptual contributions that improve the understanding of fault tolerance and state machine replication.

First, we refine the definition of fault tolerance to more accurately reflect the needs of deployed systems. Our refinement comes in two parts. First, we reject the traditional dichotomy between crash and Byzantine fault tolerance and instead embrace the Upright failure model (Chapter 2). Embracing the UpRight model allows system developers to ask "Do I want fault tolerance or not?" rather than decide in advance whether Byzantine or crash fault tolerance is appropriate for the deploy-

ment environments. Second, we reject the exclusive focus on best-case performance and observe that fault tolerant systems should provide good performance even when failures occur (Chapter 3).

Second, we clarify the definition of state machine replication. We refine the responsibilities of the replication library and the application (Chapter 4) and revisit the key functional pieces of state machine replication (Chapter 5). With respect to the responsibilities of the library and the application, we emphasize that the library is responsible for delivering batches of requests to application replicas in a single order. The application replicas are in turn responsible for executing those batches deterministically and providing, on-demand, determinist checkpoints of their state. With respect to the functional pieces of state machine replication, we observe that request authentication must be added to the traditional steps of order, agree, and execute[1].

Third, we clarify the design of replication protocols around variations of consensus (Chapter 6). By mapping the interactions between nodes in the system to a consensus problem we are able to better understand the requirements of each component of the system and leverage the existing body of work on consensus. Moving forward, the recognition that state machine replication can be described as a collection of consensus protocols should make it easier to understand new and existing protocols and also highlights the fundamental differences between systems.

While we have made it easier to understand and deploy Byzantine fault tolerant systems, there are still significant barriers to wide spread adoption. Chief among these barriers to adoption is the widespread belief that "Byzantine failures just don't happen." If true, this implies that Byzantine fault tolerant systems are a luxury that is not needed in a general computing environment. The next round of Byzantine fault tolerant systems research consequently should focus on deployment, failure tracking, and failure analysis. The key questions to answer are (a) what fraction of failures can be masked by BFT techniques and not CFT techniques and (b) what is the real impact of these failures.

[1]Note that order and agree are frequently merged into a single step.

Appendix A

UpRight Library Byte Specifications

This Appendix provides the full byte definition for all data structures that are sent across the network or placed on disk in the UpRight library. This appendix provides the byte specification for the inter-stage messages, order stage checkpoints, execution stage checkpoints, and intra-execution stage messages. We do not include the byte specification for intra-order stage messages.

A.1 Basic Message Structure

All messages in the UpRight library conform to the basic structure shown in Figure A.1[1]. Every message contains (a) a 2 byte message tag, (b) a 4 byte payload size, (c) a payload of the specified size, and (d) a block of bytes dedicated to authentication as shown in Figure A.1.

We implement three distinct authentication strategies: (1) simple MAC authentication, (2) MAC authenticator authentication, and (3) matrix signature [3] authentication.

We use MD5 for digests/hashes and SHA1 for MAC authentication.

In our current implementation, an individual MAC is 16 bytes and a digest is 20 bytes. For subsequent message definitions we will indicate which of the au-

[1]The fields in Figure A.1 and all other figures in this chapter are presented in the order they appear. The sizes of fields in the figures do not correlate with the byte size of the implementations.

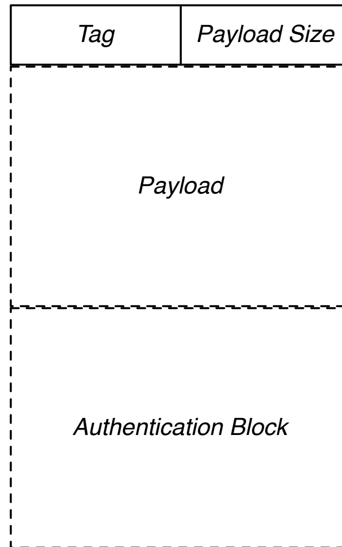

Figure A.1: Messages are built upon a verified message base. This basis byte structure contains 4 fields: tag, payload size, payload, authentication

thentication types are being used and describe the byte specification for the payload of the specific message.

Simple MAC authentication A MAC is a shared private key between a pair of nodes. Authenticating a MAC ensures that one of the nodes sharing that key generated the message. Messages authenticated with a MAC follow the structure shown in Figure A.2. The authentication block of MAC messages contains a 4 byte sender field and a 20 byte MAC. The MAC is computed over the tag, payload size, payload, and sender fields of the message.

MAC authenticator authentication A MAC authenticator [18] is an array of MACs designed to provide authentication to multiple recipients. The byte layout of a MAC authenticator message is shown in Figure A.3. The authentication block of a MAC authenticator message consists of (a) a 4 byte sender field, (b) a 16 byte digest of the tag, payload size, payload, and sender fields, and (c) one 20 byte MAC per recipient. The digest is computed over the tag, payload size, payload and sender fields. For efficiency, the MACs are computed over the digest.

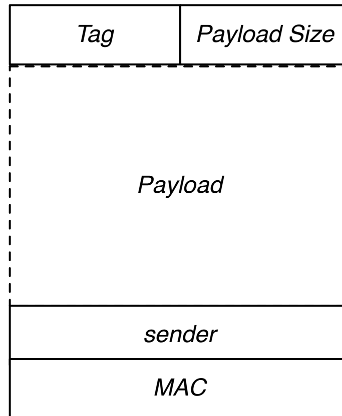

Figure A.2: Basic byte structure of a message with simple MAC authentication.

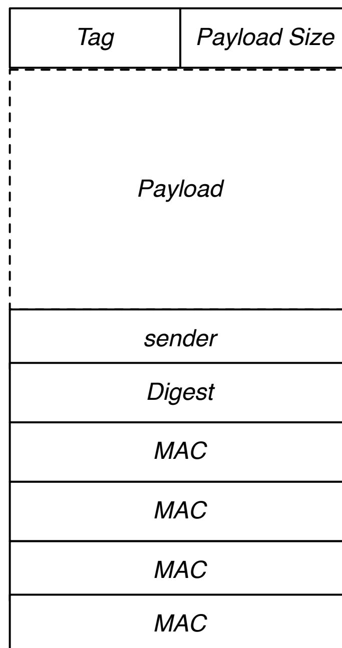

Figure A.3: Byte definition for a message authenticated with a MAC array. The sender is the replica responsible for generating the MACs, the Digest field is a digest of the tag, payload size, and sender fields. The MACs are generated using the byte representation of the digest rather than the full message.

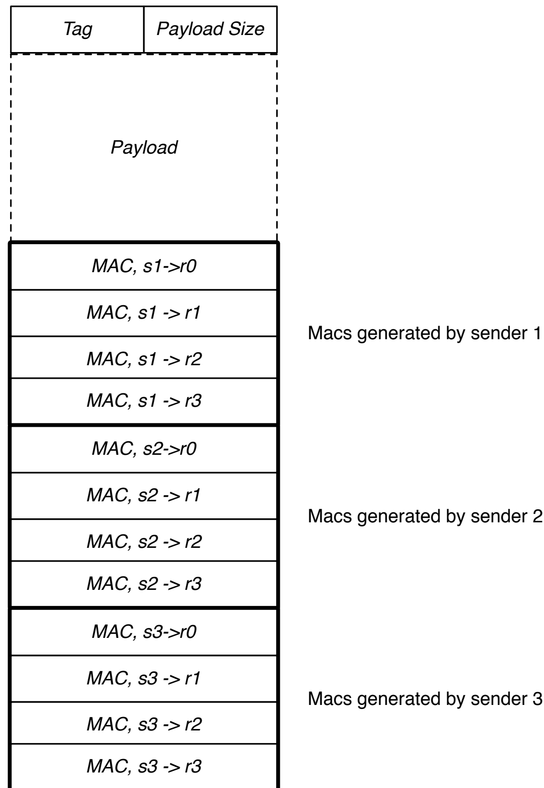

Figure A.4: Message authenticated with a matrix signature. The authentiation block of these messages consists of a collection of MAC Arrays that each authenticate the tag, size and payload.

Matrix signature authentication Matrix signatures [3] are a technique that provide the strong properties of digital signatures (specifically forwardability) at the lower costs afforded by MACs. A matrix signature consists of a collection of MAC authenticators from muliple senders. A recipient considers a matrix signature valid if it can authenticate a threshold of th MAC authenticators. The byte layout of a matrix signature message is shown in Figure A.4. The authentication block of a matrix signature message consists of (a) a 16 byte digest of the tag, payload size, and payload fields, (b) followed by k MAC authenticators with |recipient set| 20 byte MACs each. For efficiency, the individual MACs are computed over a digest of the tag, payload size, and payload fields of the message.

Message	Tag
$\langle \text{CLIENT-REQ}, \langle \text{REQ-CORE}, c, n_c, \text{OP} \rangle, c \rangle_{\vec{\mu}_{c,\mathcal{F}}}$	1 (regular) 16 (read only)
$\langle \text{AUTH-REQ}, \langle \text{REQ-CORE}, c, n_c, \text{hash}(\text{OP}) \rangle_{\vec{\mu}_{f,\mathcal{O}}}, f \rangle_{\vec{\mu}_{f,\mathcal{O}}}$	19
$\langle \text{COMMAND}, n_o, c, n_c, \text{OP}, f \rangle_{\mu_{f,e}}$	22
$\langle \text{toCache}, c, n_c, \text{OP}, f \rangle_{\vec{\mu}_{f,\mathcal{E}}}$	25
$\langle \text{NEXT-BATCH}, v, n_o, \mathcal{H}, \mathcal{B}, t, \text{bool}, o \rangle_{\vec{\mu}_{o,\mathcal{E}}}$	11 (speculative) 12 (tentative) 13 (committed)
$\langle \text{REQUEST-CP}, n_o, o \rangle_{\vec{\mu}_{o,\mathcal{E}}}$	10
$\langle \text{RETRANSMIT}, c, n_o, o \rangle_{\vec{\mu}_{o,\mathcal{E}}}$	4
$\langle \text{LOAD-CP}, \mathcal{T}_{\text{cp}}, n_o, o \rangle_{\mu_{o,e}}$	5
$\langle \text{BATCH-COMPLETE}, v, n_o, \mathcal{C}, e \rangle_{\vec{\mu}_{e,\mathcal{F}}}$	20
$\langle \text{FETCH}, n_o, c, n_c, \text{hash}(\text{OP}), e \rangle_{\vec{\mu}_{e,\mathcal{F}}}$	21
$\langle \text{CP-UP}, n_o, \mathcal{C}, e \rangle_{\vec{\mu}_{e,\mathcal{F}}}$	24
$\langle \text{LAST-EXEC}, n_e, e \rangle_{\vec{\mu}_{e,\mathcal{O}}}$	6
$\langle \text{CP-TOKEN}, n_o, \mathcal{T}_{\text{cp}}, e \rangle_{\vec{\mu}_{e,\mathcal{O}}}$	7
$\langle \text{CP-LOADED}, n_o, e \rangle_{\vec{\mu}_{e,\mathcal{O}}}$	14
$\langle \text{REPLY}, n_c, \mathcal{R}, \mathcal{H}, e, \rangle_{\mu_{e,c}}$	8 (regular) 15 (watch) 17 (readonly)

Table A.1: Message Tags for all intra-node messages.

A.2 Inter-stage messages

This section defines the byte specification and message tags for all messages exchanged betweeen clients, filter, order, and execution nodes. Details on messages that are internal to the execution stage can be found in Sections A.3.

A.2.1 Message Tags

The specific message tags used for all inter-node messages are shown in Table A.1.

A.2.2 Inter-stage messages

This section defines the payload structure for all messages that pertain directly to client requests.

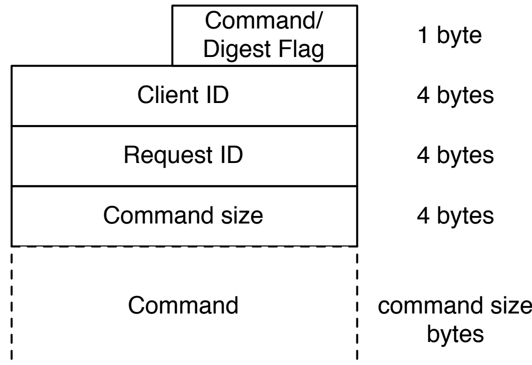

Figure A.5: Byte Specification of the Entry at the core of every request.

Entry. All messages which contain a request in their payload are built around a common *entry* data structure shown in Figure A.5. An entry consists of five fields: (1) a 1 byte flag indicating if the entry contains a command or a digest of a command, (2) a 4 byte identifier of the client that issued the command, (3) a 4 byte request identifier, (4) the size, in bytes, of the command (or command digest), and (5) the command (or digest) itself.

Client Requests. A $\langle \text{CLIENT-REQ}, \langle \text{REQ-CORE}, c, n_c, \text{OP} \rangle, c \rangle_{\vec{\mu}_{c,\mathcal{F}}}$ message relies on the MAC authenticator byte layout. The payload of the message is an entry shown in Figure A.5.

Filtered Requests. A $\langle \text{AUTH-REQ}, \langle \text{REQ-CORE}, c, n_c, \text{hash}(\text{OP}) \rangle_{\vec{\mu}_{f,\mathcal{O}}}, f \rangle_{\mu_{f,o}}$ message relies on the MAC authenticator byte layout. The implementation of each filtered request message contains one or more requests that have been individually validated by the sending filter replica. The payload of a filtered request message is a 2 byte integer k followed by k authenticated entries. An authenticated entry is a message authenticated by a matrix signature whose payload is an entry. The payload of a filtered request message is shown in Figure A.6.

Forward Requests. A $\langle \text{COMMAND}, n_o, c, n_c, \text{OP}, f \rangle_{\mu_{f,e}}$ message relies on MAC authentication. The payload of a forward request message is a 4 byte sequence number followed by a request entry. The entry in a forward request is always a command

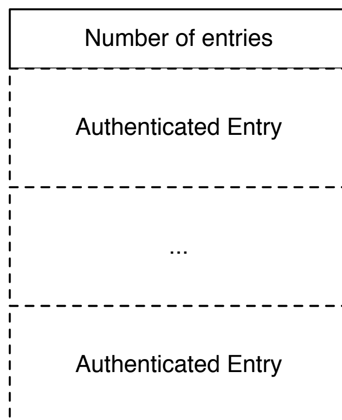

Figure A.6: Byte Specification of the payload of a $\langle$AUTH-REQ, $\langle$REQ-CORE, c, n_c, hash(OP)$\rangle_{\vec{\mu}_{f,\mathcal{O}}}$, $f\rangle_{\mu_{f,o}}$ message.

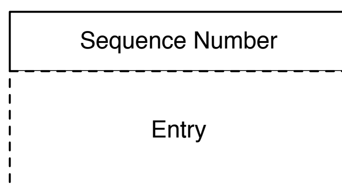

Figure A.7: Byte Specification of the payload of a $\langle \text{COMMAND}, n_o, c, n_c, \text{OP}, f\rangle_{\mu_{f,e}}$ message.

itself and not a digest. The byte format is shown in Figure A.7

Speculatively Forwarded Requests. A $\langle \text{TOCACHE}, c, n_c, \text{OP}, f\rangle_{\vec{\mu}_{f,\mathcal{E}}}$ messages uses the MAC authenticator authentication byte layout. The payload of this message is a request entry shown in Figure A.5.

Next Batch. A $\langle \text{NEXT-BATCH}, v, n_o, \mathcal{H}, \mathcal{B}, t, \text{bool}, o\rangle_{\vec{\mu}_{o,\mathcal{E}}}$ message uses the MAC authenticator byte layout. The payload of the message is shown in Figure A.8 and consists of 9 fields: (1) a 4 byte view number, (2) a 4 byte sequence number, (3) a 16 byte history digest, (4) a 16 byte checkpoint digest, (5) a 2 byte boolean, (6) a 2 byte integer for the byte size of encoding non-determinism and time, (7) encoded non-determinism and time, (8) a 2 byte integer representing the number of commands

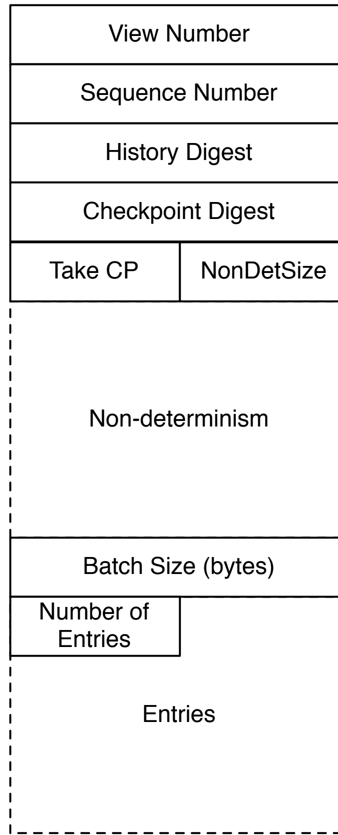

Figure A.8: Byte Specification of a $\langle \text{NEXT-BATCH}, v, n_o, \mathcal{H}, \mathcal{B}, t, \text{bool}, o\rangle_{\vec{\mu}_{o,\mathcal{E}}}$ message

in the batch, and (9) an entry per command in the bath. Non-determinism is encoded as a pair of 8 byte numbers corresponding to time and a seed for a pseudo random number generator as shown in Figure A.9. There are three different types of NextBatch messages corresponding to the level of agreement achieved by the order node: speculative, tentative, and commmitted.

Replies A $\langle \text{REPLY}, n_c, \mathcal{R}, \mathcal{H}, e, \rangle_{\mu_{e,c}}$ message is based on the simple MAC authentication byte layout. The payload of a reply consists of (a) a 4 byte sequence number, (b) a 4 byte encoding of the size of the reply, (c) and the reply itself as shown in Figure A.10.

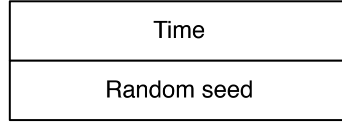

Figure A.9: Byte encoding of non-determinism. The two fields correspond to time and a seed for random number generation.

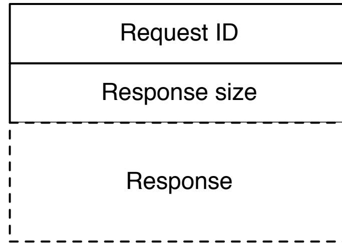

Figure A.10: Byte Specification of the $\langle \text{REPLY}, n_c, \mathcal{R}, \mathcal{H}, e, \rangle_{\mu_{e,c}}$ message.

Checkpoint request. A $\langle \text{REQUEST-CP}, n_o, o\rangle_{\vec{\mu}_{o,\mathcal{E}}}$ message relies on the MAC authenticator byte layout. The payload of a checkpoint request consists of a 4 byte sequence number for the checkpoint being requested as shown in Figure A.11.

Checkpoint release. A $\langle \text{RELEASE-CP}, \mathcal{T}_{\text{cp}}, n_o, o\rangle_{\vec{\mu}_{o,\mathcal{E}}}$ message relies on the MAC authenticator byte layout. The payload of a checkpoint release message consists of (a) a four byte sequence number of the checkpoint to be released, (b) a four byte length of the checkpoint token, and (c) the checkpoint token itself as shown in Figure A.12.

Retransmit. A $\langle \text{RETRANSMIT}, c, n_o, o\rangle_{\vec{\mu}_{o,\mathcal{E}}}$ message relies on the MAC authenticator byte layout. The payload of a retransmit message consists of (a) a 4 byte

Figure A.11: Byte Specification of the payload for a $\langle \text{REQUEST-CP}, n_o, o\rangle_{\vec{\mu}_{o,\mathcal{E}}}$ message.

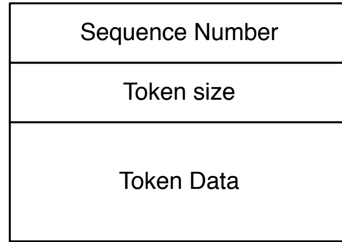

Figure A.12: Byte Specification of the payload for a $\langle \text{RELEASE-CP}, \mathcal{T}_{\text{cp}}, n_o, o \rangle_{\vec{\mu}_{o,\mathcal{E}}}$ message.

Figure A.13: Byte Specification of the payload for a $\langle \text{RETRANSMIT}, c, o, \vec{\mu}_{o,\mathcal{E}} \rangle_m$essage.

client identifier and (b) a 4 byte batch identifier as shown in Figure A.13.

Load checkpoint. A $\langle \text{LOAD-CP}, \mathcal{T}_{\text{cp}}, n_o, o \rangle_{\mu_{o,e}}$ message uses the simple MAC authentication byte layout. The fields of a load checkpoint message are (a) a 4 byte sequence number, (b) a 4 byte length of a checkpoint descriptor, and (c) the checkpoint to be loaded as shown in Figure A.14.

Batch Completed. A $\langle \text{BATCH-COMPLETE}, v, n_o, \mathcal{C}, e \rangle_{\vec{\mu}_{e,\mathcal{F}}}$ message relies on the MAC authenticator byte layout. The payload consists of (a) a 4 byte view number,

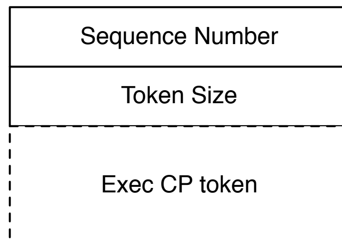

Figure A.14: Byte Specification of the payload for a $\langle \text{LOAD-CP}, \mathcal{T}_{\text{cp}}, n_o, o \rangle_{\mu_{o,e}}$ message.

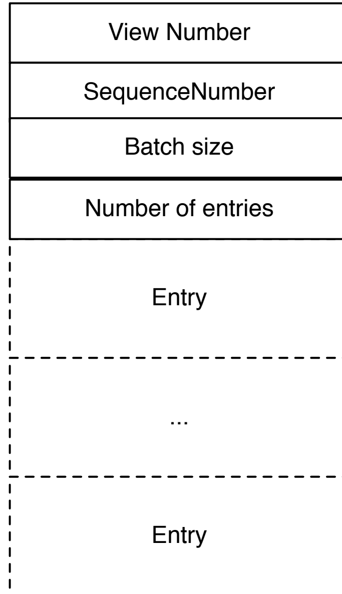

Figure A.15: Byte specification of a $\langle \text{BATCH-COMPLETE}, v, n_o, \mathcal{C}, e \rangle_{\vec{\mu}_{e,\mathcal{F}}}$ message.

(b) a 4 byte sequence number, (c) a 4 byte count of the subsequent bytes in the payload, (d) a 2 byte count k of the number of contained entries, and (e) k entries as shown in Figure A.15. The byte layout of each entry is shown in Figure A.5.

Fetch Command. A $\langle \text{FETCH}, n_o, c, n_c, \text{hash}(\text{OP}), e \rangle_{\vec{\mu}_{e,\mathcal{F}}}$ message relies on the MAC authenticator byte layout. The payload consists of a 4 byte sequence number and an entry as shown in Figure A.16. The byte layout of the entry is shown in Figure A.5.

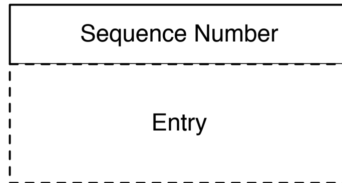

Figure A.16: Byte specification of a $\langle \text{FETCH}, n_o, c, n_c, \text{hash}(\text{OP}), e \rangle_{\vec{\mu}_{e,\mathcal{F}}}$ message.

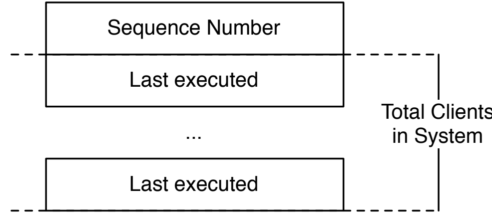

Figure A.17: Byte specification of a $\langle \text{CP-UP}, n_o, \mathcal{C}, e \rangle_{\vec{\mu}_{e,\mathcal{F}}}$ message.

Figure A.18: Byte Specification of $\langle \text{LAST-EXEC}, n_e, e \rangle_{\vec{\mu}_{e,\mathcal{O}}}$ and $\langle \text{CP-LOADED}, n_o, e \rangle_{\vec{\mu}_{e,\mathcal{O}}}$ messages.

Checkpoint Update. A $\langle \text{CP-UP}, n_o, \mathcal{C}, e \rangle_{\vec{\mu}_{e,\mathcal{F}}}$ message relies on the MAC authenticator byte layout. The payload consists of (a) a 4 byte sequence number and (b) the 4 byte identifier of the most recent request executed for each client as shown in Figure A.17. The checkpoint update message is used in place of a batch completed message when the execution stage processes retransmission instructions.

Last executed. A $\langle \text{LAST-EXEC}, n_e, e \rangle_{\vec{\mu}_{e,\mathcal{O}}}$ message relies on the MAC authenticator byte layout. The payload of a last executed message is a 4 byte sequence number shown in Figure A.18.

Checkpoint loaded. A $\langle \text{CP-LOADED}, n_o, e \rangle_{\vec{\mu}_{e,\mathcal{O}}}$ uses the MAC authenticator byte layout. The payload consist of a 4 byte sequence number shown in Figure A.18. Note that the checkpoint loaded message is used as an efficient replacement for a last executed message. The handling of the two messages is identical, except that instructions to load a checkpoint are never sent in response to a checkpoint loaded notification.

Checkpoint message. A $\langle \text{CP-TOKEN}, n_o, \mathcal{T}_{\text{cp}}, e \rangle_{\vec{\mu}_{e,\mathcal{O}}}$ message relies on the MAC authenticator byte layout. The payload of the message consists of (a) a 4 byte sequence number, (b) the size of the checkpoint, and (c) the checkpoint as shown in

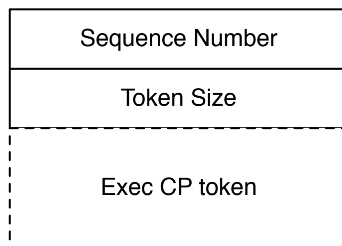

Figure A.19: Byte specification for the payload of a $\langle \text{CP-TOKEN}, n_o, \mathcal{T}_{\text{cp}}, e\rangle_{\vec{\mu}_{e,\mathcal{O}}}$ message.

Message	Tag
$\langle \text{FETCH-EXEC-CP}, n, e\rangle_{\vec{\mu}_{e,\mathcal{E}}}$	70
$\langle \text{EXEC-CP-STATE}, n, \mathcal{S}, e\rangle_{\mu_{e,e'}}$	71
$\langle \text{FETCH-STATE}, \mathcal{T}_{\text{state}}, e\rangle_{\vec{\mu}_{e,\mathcal{E}}}$	72
$\langle \text{STATE}, \mathcal{T}_{\text{state}}, \mathcal{S}, e\rangle_{\mu_{e,e'}}$	73

Table A.2: Set of messages for intra-node communication

Figure A.19.

A.2.3 Order stage checkpoint

During normal operation, the order stage periodically records checkpoints to disk. The contents of a checkpoint are shown in Figure A.20. The basic layout of bytes when serializing an order checkpoint is shown in Figure A.21. The serialization consists of (a) a 16 bytes history digest, (b) an 8 byte time, (c) a 4 byte sequence number, (d) a 2 byte count of the number of clients k in the system, (e) k pairs of 4 byte request identifiers and 4 byte sequence numbers, and (f) the execution checkpoint.

A.3 Execution node specifications

A.3.1 Message Tags

The message tags used for all intra-execution messages are shown in Table A.2.

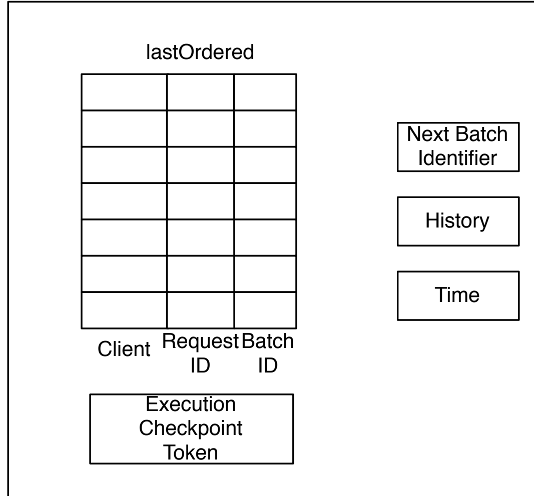

Figure A.20: Order node checkpoint.

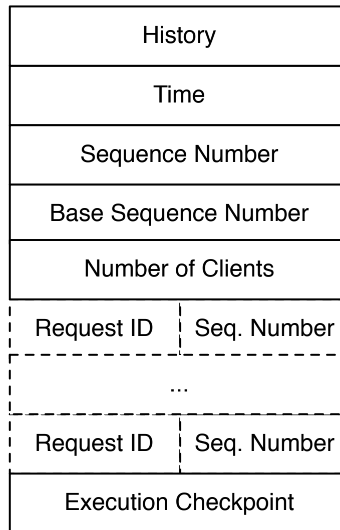

Figure A.21: Order node checkpoint byte specification.

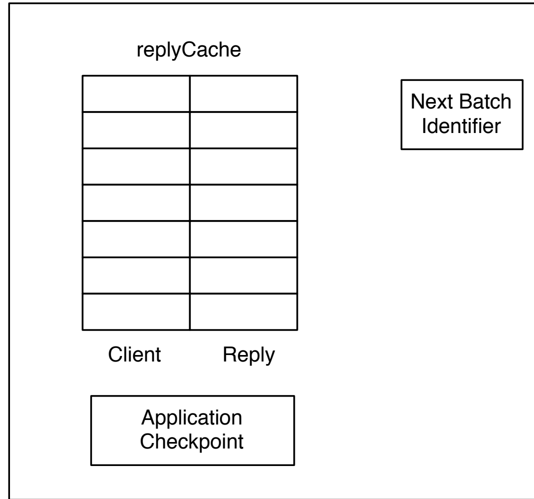

Figure A.22: Exec node checkpoint.

A.3.2 Execution checkpoints

During normal operation, order nodes record checkpoints to disk. The contents of a checkpoint are shown in Figure A.22. The basic layout of bytes when serializing an order checkpoint is shown in Figure A.23. The serialization consists of (a) a 4 byte base sequence number, (b) a 4 byte current sequence number, (c) a 4 byte max sequence number, (d) a 4 byte size of the application checkpoint, (e) the application checkpoint, and (f) for each client, (i) a 4 byte sequence number, (ii) a 4 byte request id, (iii) a 4 byte response length k, and (iv) a REPLY message. The serialization is shown in Figure A.23.

A.3.3 Execution Messages

Fetch checkpoint. A $\langle \text{FETCH-EXEC-CP}, n, e\rangle_{\vec{\mu}_{e,\mathcal{E}}}$ message relies on the MAC authenticator byte layout. The paylod consists of a 4 byte sequene number shown in Figure A.24.

Execution Checkpoint. A $\langle \text{EXEC-CP-STATE}, n, \mathcal{S}, e\rangle_{\mu_{e,e'}}$ message uses the basic MAC authentication byte layout. The payload consists of a 4 byte sequence number, a 4 byte checkpoint size, and the checkpoint from Figure A.23. The byte layout of

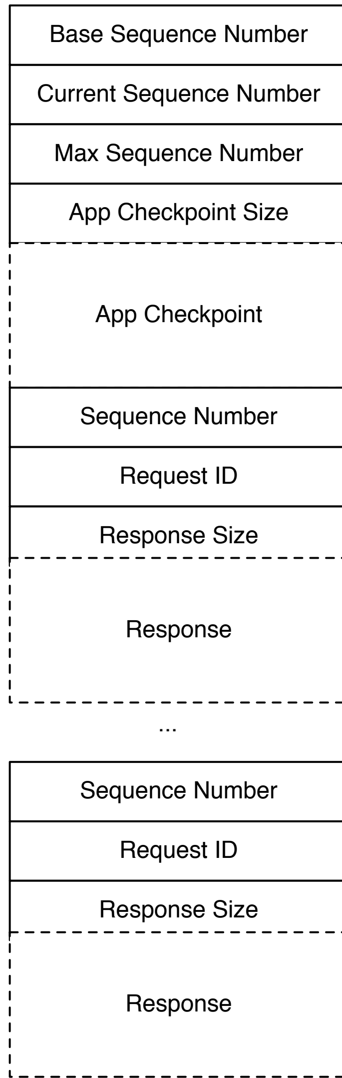

Figure A.23: Order node checkpoint byte specification.

Figure A.24: Byte Specification of the payload of a $\langle \textsc{fetch-exec-cp}, n, e \rangle_{\vec{\mu}_{e,\mathcal{E}}}$ message.

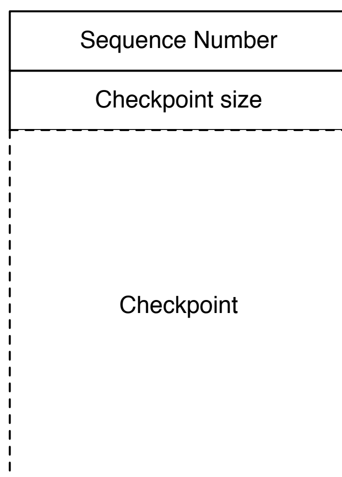

Figure A.25: Byte Specification of the payload of a $\langle \text{EXEC-CP-STATE}, n, \mathcal{S}, e \rangle_{\mu_{e,e'}}$ message.

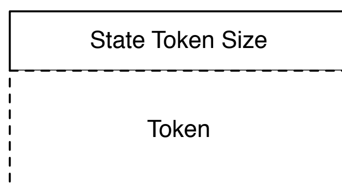

Figure A.26: Byte Specification of the payload of a $\langle \text{FETCH-STATE}, \mathcal{T}_{\text{state}}, e \rangle_{\vec{\mu}_{e,\mathcal{E}}}$ message.

the payload is shown in Figure A.25.

Fetch State. A $\langle \text{FETCH-STATE}, \mathcal{T}_{\text{state}}, e \rangle_{\vec{\mu}_{e,\mathcal{E}}}$ message uses the MAC authenticator byte layout. The payload consists of a 2 byte token size and a token as shown in Figure A.26.

Send State. A $\langle \text{STATE}, \mathcal{T}_{\text{state}}, \mathcal{S}, e \rangle_{\mu_{e,e'}}$ message relies on the simple MAC authentication byte layout. The payload consists of a 2 byte token size, the token, a 4 byte state size, and then the state as shown in Figure A.27.

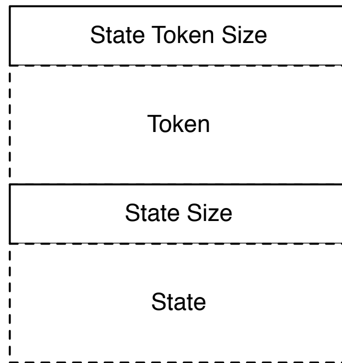

Figure A.27: Byte Specification of the payload of a $\langle \text{STATE}, \mathcal{T}_{\text{state}}, \mathcal{S}, e \rangle_{\mu_{e,e'}}$ message.

Appendix B

UpRight Library API

This appendix describes the Java API between the UpRight library and replicated applications. Section B.1 describes the application client-library client API. Section B.2 describes the application server-library server API.

B.1 Client API

Figure B.1 depicts the four function calls provided by the UpRight library to application clients. The function calls provide synchronous (*execute*) and asynchronous (*enqueue*) calls instructing the library to execute general or read-only requests. Each call takes a byte array representation of the application request as a parameter.

Figure B.2 depicts the functions that the application client should implement. All three functions are optional and are not required to support basic functionality of synchronous request execution. The function *brokenConnection* is used to signal the application when a network error occurs and is important for applications that explicitly rely on TCP connections to maintain sessions. The function *returnReply* is used in conjunction with asynchronous request execution and server initiated communication. The function *canonicalEntry* allows the application to select a canonical response (if it exists) from a quorum of responses that are semantically equivalent but based on different byte representations.

```
1   /**
2     Returns the result of executing operation through the normal execution path
3   **/
4    public byte[] execute(byte[] operation);

6   /**
7     Returns the result of execution read only request operation.
8   **/
9    public byte[] executeReadOnlyRequest(byte[] operation);

11  /**
12    Enqueues a read only request for asynchronous execution
13  **/
14  public void enqueueReadOnlyRequest(byte[] op);

16  /**
17    Enque a regular request for asynchronous execution
18  **/
19  public void enqueueRequest(byte[] operation);
```

Figure B.1: Interface exported by the UpRight library to the application client.

```
1      /**
2           Function called when the connection between the client and the
3           server is determined to be broken.
4        **/
5       public void brokenConnection();

7       /**
8          Returns a reply received from the server
9        **/
10      public void returnReply(byte[] reply);

12      /**
13         Considers the set of possible replies options.  Returns a
14         canonical version of those replies if it exists, returns null
15         otherwise
16       **/
17      public byte[] canonicalEntry(byte[][] options);
```

Figure B.2: Interface implemented by the application client.

B.2 Server API

Figure B.3 shows the six functions an UpRight application must impement.

The first two functions, *execute* and *executeReadOnly*, are used to execute ordered batches or requests and read only requests respectively. The application calls *result* once per executed request. The application is expected to execute batches in the order they are received using the time and PRNG in the nondeterminism field. The final parameter of *execute* is a boolean *takeCP*. If *takeCP* is true, then the application is expected to take a checkpoint immediately after executing the current batch and before processing any requests contained in subsequent batches.

The second two functions, *loadCP* and *releaseCP*, are required for managing application checkpoints. The function *loadCP* takes a byte array describing either the application checkpoint, or a description that the application can use to map to a specific checkpoint, and instructs the application to load the specified checkpoint. The function *releaseCP* is an optional call that should be used if the application provides a descriptor of its checkpoint rather than the checkpoint itself. *releaseCP* allows the application to manage checkpoints internally and potentially rely on incremental checkpoints for efficiency.

The final two functions, *fetchState* and *loadState* are used in the case that the application reports tokens describing a checkpoint to the library. These functions are used to transfer small portions of an application checkpoint between replicas and allow for incremental state transfer when checkpoints are loaded.

Figure B.4 shows the five functions exported by the UpRight library to the application server. The function *result* is called whenever the application finishes processing a request and inidcates the *result* of that computation, the *clientId* of the client that issued the request, the *reqId* associated with the request, and a boolen *toCache* indicating if the request is a regular response (*true*) and should be stored in the reply cache or not. The function *readOnlyResult* serves the same purpose, but is used only for responses to read only requests.

The function *returnCP* is called by the application when it finishes generating a checkpoint following the execution of batch *seqno*. The checkpoint is described by the byte array *AppCPToken*; the format of *AppCPToken* is dictated by the application.

The functions *returnState* and *requestState* facilitate the transfer of applica-

```
1   /**********************
2   Request Execution
3   **********************/
4         /**
5           Execute the commands in batch with associated order sequence
6           number seqNo and using time for any non-determinism

8           Following each command in batch, shim.result() is called.
9          **/
10        public void exec(CommandBatch batch,  long seqNo,
11                         NonDeterminism time, boolean takeCP);

13        /**
14          Execute operation as a read only request.

16          Following execution of operation, shim.readOnlyResult() is
17          called.
18         **/
19         public void execReadOnly(int clientId, long reqId,  byte[] operation);



23  /**********************
24  Checkpoint Management
25  **********************/
26        /**
27          Load the application checkpoint indicated by cpToken

29          Returns true if the checkpoint is successfully loaded,
30          returns false otherwise

32          When loadCP returns, it indicates that any requests executed as
33          part of a preceding call to exec() or execReadOnly() that have
34          not already generated a response will not generate a future
35          response.
36         **/
37        public void loadCP(byte[] appCPToken, long seqNo);

39        /**
40           Release the application checkpoint described by appCPToken.
41         **/
42        public void releaseCP(byte[] appCPToken);



46  /**********************
47  State Transfer
48  **********************/
49        /**
50           Fetch the state described by stateToken.
51         **/
52        public void fetchState(byte[] stateToken);

54        /**
55           Load the state state that is described by stateToken.
56         **/
57        public void loadState(byte[] stateToken, byte[] state);
```

Figure B.3: Interface implemented by the application server and called by the UpRight library. The six functions can be considered as three pairs of common functionality: (a) request execution, (b) checkpoint management, and (c) state transfer.

```
1   /*********************
2   Request Execution
3   *********************/
4   /**
5       Upcall that delivers the result of executing clientId's
6       reqId^th request at seqNo position in the sequence to the shim.
7   **/
8   public void result(byte[] result, int clientId, long reqId,
9                              long seqNo, boolean toCache);

11  /**
12      Upcall delivering the result of executing clientId's reqId^th
13      read only request.
14   **/
15  public void readOnlyResult(byte[] result, int clientId, long reqId);


18  /*********************
19  Checkpoint Management
20  *********************/
21   /**
22      Upcall delivering the Application checkpoint token cpToken
23      taken at batch number seqNo to the shim
24   **/
25  public void returnCP(byte[] AppCPToken,  long seqNo);


28  /*********************
29  State Transfer
30  *********************/
31  /**
32     Upcall delivering the application state corresponding to
33     stateToken to the shim
34  **/
35  public void returnState(byte[] stateToken, byte[] state);

37  /**
38     Upcall request application state described by stateToken from
39     the shim
40  **/
41  public void requestState(byte[] stateToken);
```

Figure B.4: Interface exported by the UpRight library to the application server as call-backs. The functions can be considered in groups based on common functionality: (a) response processing, (b) checkpoint management, (c) state transfer, and (d) generic management.

tion state between execution replicas. The application is expected to call *returnState* as part of processing a *fetchState* command. The application is expected to call *requestState* if it does not have a copy of the state required to successfully load a checkpoint.

Bibliography

[1] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proc. 20th SOSP*, Oct. 2005.

[2] T. Abdollah. LAX outage is blamed on 1 computer. *Los Angeles Times*, Aug. 2007.

[3] A. S. Aiyer, L. Alvisi, R. Bazzi, and A. Clement. Matrix signatures: From MACs to digital signatures in distributed systems. In *Proc. DISC 2008)*, pages 16–31, Oct. 2008.

[4] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth. BAR fault tolerance for cooperative services. In *Proc. 20th SOSP*, Oct. 2005.

[5] L. Alvisi and K. Marzullo. Message Logging: Pessimistic, Optimistic, Causal, and Optimal. *IEEE Transactions on Software Engineering*, 24(2):149–159, February 1998.

[6] Amazon elastic compute cloud. http://aws.amazon.com/ec2/, Mar. 2009.

[7] Y. Amir, B. Coan, J. Kirsch, and J. Lane. Byzantine replication under attack. In *International Conference on Dependable Systems and Networks*, June 2008.

[8] W. Bartlett and L. Spainhower. Commercial fault tolerance: A tale of two systems. *IEEE Transactions on Dependable and Secure Computing*, 1(1):87–96, 2004.

[9] BFT project homepage. http://www.pmg.csail.mit.edu/bft/#sw.

[10] R. H. Black. Using proven aircraft avionics principles to support a responsive space infrastructure. In *4th Responsive Space Conference*, 2006.

[11] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4):824–840, 1985.

[12] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proc. 7th OSDI*, 2006.

[13] C. Cachin, K. Kursawe, and V. Shoup. Random oracles in constantipole: practical asynchronous byzantine agreement using cryptography (extended abstract). In *PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, pages 123–132, New York, NY, USA, 2000. ACM.

[14] M. Calore. Ma.gnolia suffers major data loss, site taken offline. *Wired*, Jan. 2009.

[15] R. Canetti and T. Rabin. Optimal Asynchronous Byzantine Agreement. Technical Report 92-15, TR 92-15, Dept. of Computer Science, Hebrew University, 1992.

[16] M. Castro. *Practical Byzantine Fault Tolerance.* PhD thesis, Jan. 2001.

[17] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. 3rd OSDI*, pages 173–186, Feb. 1999.

[18] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 2002.

[19] T. Chandra, R. Griesmer, and J. Redstone. Paxos made live – an engineering perspective. In *Proc. 26th PODC*, June 2007.

[20] P. Chen, E. Lee, G. Gibson, R. Katz, and D. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys*, 26:145–185, 1994.

[21] At LAX, computer glitch delays 20,000 passengers. http://travel.latimes.com/articles/la-trw-lax12aug12.

[22] G. Chockler, D. Malkhi, and M. Reiter. Backoff protocols for distributed mutual exclusion and ordering. In *ICDCS-21*, pages 11–20, 2001.

[23] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. Upright cluster services. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 277–290, New York, NY, USA, 2009. ACM.

[24] A. Clement, M. Marchetti, E. Wong, L. Alvisi, and M. Dahlin. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Proc. 6th NSDI*, Apr. 2009.

[25] T. Condie, V. Kacholia, S. Sankararaman, J. M. Hellerstein, and P. Maniatis. Induced churn as shelter from routing-table poisoning. In *NDSS*, 2006.

[26] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *Proc. 7th OSDI*, Nov. 2006.

[27] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: from simple message diffusion to byzantine agreement. *Inf. Comput.*, 118(1):158–179, 1995.

[28] C. Delporte-Gallet, H. Fauconnier, F. C. Freiling, L. D. Penso, and A. Tielmann. From crash-stop to permanent omission: Automatic transformation and weakest failure detectors. In A. Pelc, editor, *DISC*, volume 4731 of *Lecture Notes in Computer Science*, pages 165–178. Springer, 2007.

[29] D. Dolev and H. R. Strong. Authenticated algorithms for Byzantine agreement. *Siam Journal Computing*, 12(4):656–666, Nov. 1983.

[30] A. Doudou, B. Garbinato, R. Guerraoui, and A. Schiper. Muteness Failure Detectors, Specification and Implementation. Technical report, 1999.

[31] K. Driscoll, B. Hall, M. Paulitsch, P. Zumstag, and H. Sivencrona. The real Byzantine generals. In *Digital Avionics Ssytems Conference*, 2004.

[32] P. Dutta, R. Guerraoui, and M. Vukolić. Best-case complexity of asynchronous Byzantine consensus. Technical Report EPFL/IC/200499, EPFL, Feb. 2005.

[33] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

[34] E. N. Elnozahy, L. Alvisi, Y. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[35] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[36] The FlexiProvider Group. the FlexiProvider Project. http://www.flexiprovider.de.

[37] R. Friedman and R. V. Renesse. Packing messages as a tool for boosting the performance of total ordering protocls. In *HPDC*, 1997.

[38] J. Garay and Y. Moses. Fully Polynomial Byzantine Agreement for $n>3t$ Processors in $t+1$ Rounds. *SIAM J. of Computing*, 27(1):247–290, 1998.

[39] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proc. 19th SOSP*, pages 29–43. ACM Press, 2003.

[40] T. J. Giuli, P. Maniatis, M. Baker, D. S. H. Rosenthal, and M. Roussopoulos. Attrition defenses for a peer-to-peer digital preservation system. In *USENIX*, 2005.

[41] J. Gray. A census of Tandem system availability between 1985 and 1990. *IEEE Trans. on Reliability*, 39(4), Oct. 1990.

[42] R. Guerraoui. The next 700 bft protocols. In T. P. Baker, A. Bui, and S. Tixeuil, editors, *OPODIS*, volume 5401 of *Lecture Notes in Computer Science*, page 1. Springer, 2008.

[43] Hadoop. http://hadoop.apache.org/core/.

[44] Hdfs. http://hadoop.apache.org/hdfs.

[45] J. Hendricks, G. R. Ganger, and M. K. Reiter. Low-overhead Byzantine fault-tolerant storage. In *SOSP*, 2007.

[46] M. Iacoponi. System architecture for byzantine resilient computation in launch vehicle applications. In *Digital Avionics Systems Conference*, 1990.

[47] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. The securering group communication system. *ACM Trans. Inf. Syst. Secur.*, 4(4):371–406, 2001.

[48] C. Killian, J. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *NSDI*, 2007.

[49] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *Proc. 20th SOSP*, Oct. 2007.

[50] R. Kotla and M. Dahlin. High throughput Byzantine fault tolerance. In *Conference on Dependable Systems and Networks, DSN'04*, June 2004.

[51] M. Lagos and M. Stannard. Power restored in San Francisco. *San Francisco Chronicle*, July 2007.

[52] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 1978.

[53] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[54] L. Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column)*, 32(4):51–58, Dec. 2001.

[55] L. Lamport. Lower bounds for asynchronous consensus. In *Proceedings of the International Workshop on Future Directions in Distributed Computing*, pages 22–23, June 2003.

[56] L. Lamport. Lower bounds for asynchronous consensus. Technical Report MSR-TR-2004-72, Microsoft Research, July 2004.

[57] L. Lamport. Fast Paxos. Technical Report MSR-TR-2005-112, Microsoft Research, July 2005.

[58] L. Lamport. Generalized consensus and paxos. Technical Report MSR-TR-2005-33, Mar. 2005.

[59] L. Lamport and M. Fischer. Byzantine generals and transaction commit protocols. Technical Report 62, SRI International, 1982.

[60] L. Lamport and M. Masa. Cheap paxos. In *Proc. DSN-2004*, pages 307–314, June 2004.

[61] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 1982.

[62] B. W. Lampson. Hints for computer system design. *SIGOPS Oper. Syst. Rev.*, 17, 1983.

[63] A. Mahimkar, J. Dange, V. Shmatikov, H. Vin, and Y. Zhang. dFence: Transparent network-based denial of service mitigation. In *NSDI*, 2007.

[64] D. Malkhi and M. Reiter. A high-throughput secure reliable multicast protocol. In *CSFW '96: Proceedings of the 9th IEEE workshop on Computer Security Foundations*, page 9, Washington, DC, USA, 1996. IEEE Computer Society.

[65] D. Malkhi and M. Reiter. Unreliable intrusion detection in distributed computations. In *CSFW '97: Proceedings of the 10th IEEE workshop on Computer Security Foundations*, page 116, Washington, DC, USA, 1997. IEEE Computer Society.

[66] P. Maniatis, M. Roussopoulos, T. J. Giuli, D. S. H. Rosenthal, and M. Baker. The LOCKSS peer-to-peer digital preservation system. *ACM Trans. Comput. Syst.*, 2005.

[67] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 369–384, Berkeley, CA, USA, 2008. USENIX Association.

[68] J.-P. Martin and L. Alvisi. Fast Byzantine consensus. *IEEE Transactions on Dependable and Secure Computing*, 3(3):202–215, July 2006.

[69] J.-P. Martin, L. Alvisi, and M. Dahlin. Minimal Byzantine storage. In *16th International Conference on Distributed Computing, DISC 2002*, pages 311–325, Oct. 2002.

[70] J.-P. Martin, L. Alvisi, and M. Dahlin. Small Byzantine quorum systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 02), DCC Symposium*, pages 374–383, June 2002.

[71] S. Misel. Wow, AS7007! NANOG mail archives http://www.merit.edu/mail.archives/nanog/1997-04/msg00340.html.

[72] C. Mohan, R. Strong, and S. Finkelstein. Method for distributed transaction commit and recovery using byzantine agreement within clusters of processors. In *PODC '83: Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 89–103, New York, NY, USA, 1983. ACM.

[73] J. Napper. *Robust Multithreaded Applications.* PhD thesis, The University of Texas at Austin, 2008.

[74] Netty project. http://www.jboss.org/netty.html.

[75] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn. Rethink the sync. In *Proc. 7th OSDI*, Nov. 2006.

[76] NS-2. http://www.isi.edu/nsnam/ns/.

[77] B. Oki and B. Liskov. Viewstamped replication: A general primary copy method to support highly-available distributed systems. In *Proc. 7th PODC*, 1988.

[78] D. Oppenheimer, A. Ganapathi, and D. Patterson. Why do internet services fail, and what can be done about it, 2003.

[79] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM, 27(2):228-234*, Apr. 1980.

[80] K. J. Perry and S. Toueg. Distributed agreement in the presence of processor and communication faults. *IEEE Trans. Softw. Eng.*, 12(3):477–482, 1986.

[81] E. Pinheiro, W. Weber, and L. Barroso. Failure trends in a large disk drive population. In *Proc. USENIX FAST*, 2007.

[82] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Iron file systems. In *Proc. 20th SOSP*, 2005.

[83] Query/Update protocol. http://www.pdl.cmu.edu/QU/index.html.

[84] M. K. Reiter. A secure group membership protocol. *IEEE Trans. Softw. Eng.*, 22(1):31–42, 1996.

[85] A. Rich. ZFS, sun's cutting-edge file system. Technical report, Sun Microsystems, 2006.

[86] R. Rodrigues, M. Castro, and B. Liskov. BASE: using abstraction to improve fault tolerance. In *Proc. 18th SOSP*, Oct. 2001.

[87] F. B. Schneider. Byzantine generals in action: implementing fail-stop processors. *ACM Trans. Comput. Syst.*, 2(2):145–154, 1984.

[88] F. B. Schneider. Implementing fault–tolerant services using the state machine approach: A tutorial. *Computing Surveys*, 22(3):299–319, September 1990.

[89] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, Sept. 1990.

[90] B. Schroeder and G. Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you. In *Proc. USENIX FAST*, 2007.

[91] B. Schroeder, E. Pinheiro, and W.-D. Weber. Dram errors in the wild: a large-scale field study. In *SIGMETRICS '09: Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, pages 193–204, New York, NY, USA, 2009. ACM.

[92] M. Serafini, P. Bokor, D. Dobre, M. Majuntke, and N. Suri. Scrooge: Reducing the costs of fast byzantine replication in presence of unresponsive replicas. In *Proc. of the IEEE Int'l Conf. on Dependable Systems and Networks (DSN)*, 2010.

[93] M. Serafini, P. Bokor, and N. Suri. Scrooge: Stable speculative byzantine fault tolerance using testifiers. Technical report, Darmstadt University of Technology, Department of Computer Science, September 2008.

[94] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proc. DSN-2002*, pages 389–398, 2002.

[95] A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe. Bft protocols under fire. In *NSDI'08: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 189–204, Berkeley, CA, USA, 2008. USENIX Association.

[96] Single network card downed lax computers. http://www.tgdaily.com/content/view/33398/113/.

[97] A. S. Team. Amazon S3 availability event: July 20, 2008. http://status.aws.amazon.com/s3-20080720.html, 2008.

[98] P. Thambidurai and Y.-K. Park. Interactive consistency with multiple failure modes. In *Proc. 7th SRDS*, 1988.

[99] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *Proc. 16th SOSP*, pages 224–237, 1997.

[100] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating Byzantine faults in transaction processing systems using commit barrier scheduling. In *Proc. 20th SOSP*, 2007.

[101] M. Walfish, M. Vutukuru, H. Balakrishnan, D. Karger, and S. Shenker. DDoS defense by offense. In *SIGCOMM*, 2006.

[102] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic misconfiguration troubleshooting with peerpressure. In *OSDI*, Dec. 2004.

[103] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. 5th OSDI*, pages 255–270, Boston, MA, Dec. 2002. USENIX Association.

[104] T. Wood, R. Singh, A. Venkataramani, and P. Shenoy. ZZ: Cheap practical BFT using virtualization. Technical Report TR14-08, University of Massachusetts, 2008.

[105] J. Yang, C. Sar, and D. Engler. EXPLODE: A lightweight, general system for finding serious storage system errors. In *Proc. 7th OSDI*, 2006.

[106] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 19–19, Berkeley, CA, USA, 2004. USENIX Association.

[107] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Proc. 19th SOSP*, Oct. 2003.

[108] Zookeeper. http://hadoop.apache.org/zookeeper.

Vita

Allen Clement was born in Alexandria, Virginia, as the son of two lawyers. His family moved to Houston, Texas, when he was seven and he lived there until he graduated from Strake Jesuit College Preparatory in 1996. He attended Princeton University in Princeton, New Jersy, where he graduated with an A.B. in Computer Science in 2000. He taught introductory Java programming at Ngee Ann Polytechnic in Singapore from July 2000 through June 2001. He spent fall 2001 through May 2002 studying computational geometry and hypercube embedding at the University of British Columbia in Vancouver, British Columbia. In fall 2002 he enrolled in the PhD program in the Department of Computer Sciences at the University of Texas at Austin, where he was a teaching assistant and graduate research assistant.

Permanent Address: 1902 Coulcrest
Houston, Texas, 77055

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab.