Securely Improving Performance Through Speculative Predictive Remote Execution

Bennet S. Yee

March 23, 2004

Abstract

We describe and analyze the use of a remote execution framework to implement *Mobile Code-enhanced Remote Procedure Calls* (McRPCs). This uses mobile code to implement a variation of remote procedure call (RPC) which enhances performance for some applications that are latency-bound.

The McRPC remote execution framework includes relatively straightforward modifications to standard RPC servers and the addition of untrusted remote execution engines. The untrusted remote execution service permits mobile code to use its CPU, memory, and network in a restricted manner; no accesses to other local resources are permitted. Nevertheless, this sandbox allows latency-bound computations to improve performance by using McRPCs in lieu of standard RPCs.

The use of McRPCs trades privacy of the client computation and total resource usage for decreased time-to-completion. While privacy may be lost, no compromise is made on the correctness of the computed results.

1 Introduction

Process technology improvements give us faster processors and greater bandwidth, making all computations faster. For those distributed computations that require many communications roundtrips between clients and servers, however, the physical separation between clients and servers induces a lower-bound on completion time arising from the communications delay. Unless we can move the client computation closer to the server (or vice versa), this delay appears inevitable.

If we could remotely run the client code at or near the server, the latency component of the run-time largely disappears. One important question is the security of remote execution. While we may trust the local client and the remote RPC server, we might not feel the same way about a remote execution engine. In this case, verifying the correctness of remote executions is needed. Though checking the results of RPCs is theoretically possible [1], the method is as yet impractical for use in real systems.

This paper examines the use of rollback and mobile code to improve performance. The mobile code runs on an untrusted remote execution server and we use it to predict client requests. The RPC server optimistically processes these requests, but maintains a small amount of logs to permit rollback. The user trusts the RPC server and the local client as in the standard RPC model. The remote execution engine will be able to obtain details about the client-side computation, but as I show below, will be unable to tamper with it.

I call this scheme *Mobile Code-enhanced Remote Procedure Calls* or McRPCs. The McRPC idea takes the notion of using speculative execution to prefetch data [2, 3] and uses it in the context of remote execution and mobile code.

2 McRPCs

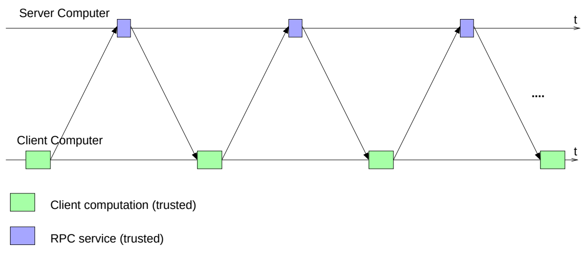


Figure 1: RPC execution time
The horizontal axis is time. The width of the boxes on the client time line indicate computation between RPCs, and the width of the boxes on the server time line indicate RPC processing. Here each RPC request/response message travels from the client computer to the server computer, incurring a distance-induced delay each time. The network message latencies dominate the job's time-to-completion.

Due to space limitations, I will make the case for

McRPCs in pictures. While Figure 1 may exaggerate communications delays relative to processing time, it shows how improving CPU power and communications bandwidth will inevitably lead to a situation where latency becomes the bottleneck. If we could migrate the client code to a trusted server that is physically close to the RPC server, Figure 2 shows the improved communcations performance.

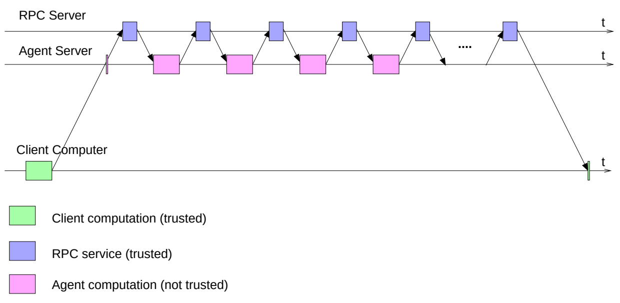


Figure 2: Mobile Agents Hide Latency

We can see graphically that the number of boxes (representing useful computation) per unit time is higher. By using mobile code to migrate closer to the remote resource, we eliminate much of the overhead arising from communications latency.

Of course, if the execution engine in Figure 2 is corrupted, it might not be possible to detect this, and even if we could, to recover to a safe state. Instead, we use the strategy in Figure 3: the requests generated by the migrated copy of the client program—which we will call the "shadow client"—is checked at every request by the local copy: the McRPC server includes a copy of the original request (or a one-way hash of it) in its authenticated response. The client "speculates" that the shadow client will execute correctly, and that it will correctly make predictive requests to the RPC server. Note that the client program does not make normal RPCs any more—instead, it is linked against an ABI-compatible McRPC library which transparently performs the remote fork and coordinates the future McRPC communications, as well as managing the cryptographic keys needed to authenticate the McRPC server's responses. The local client executes in lock-step with the remotely forked shadow client. In the absence of security attacks—and assuming deterministic execution—the request generated by the local client and the shadow client will be identical.

In Figure 3, the arrow from the client to the McRPC server indicates an authenticated acknowledgement message that the shadow-generated RPC parameters were correct. When the McRPC server receives this acknowledgement, it is free to discard log records that would enable it to roll back to its state prior to the just-acknowledged request. (Such an acknowledgement is not strictly necessary

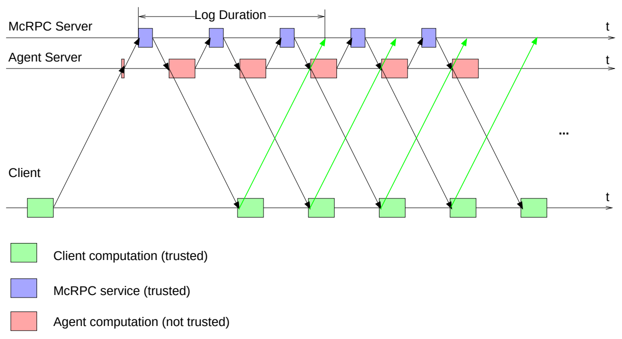


Figure 3: S^{2}PRE execution time

With McRPC, we migrate a copy of the client program to an agent server near the RPC server as before. The difference is that the migrated version is untrusted and merely makes predictive requests on the behalf of the client code, which continues to run on the client machine. Here the RPC request at the original client program need not be transmitted; instead, the matching RPC response "magically" arrives. (Assuming client and agent server computers have the same CPU speeds.)

when the request does not change the server's state.)

If, on the other hand, the client received a server response that does not correspond to its expected request, this message would contain a negative acknowledgement which requests the McRPC server to rollback. Additionally, alarms may be raised to announce that the remote execution server should no longer be trusted.

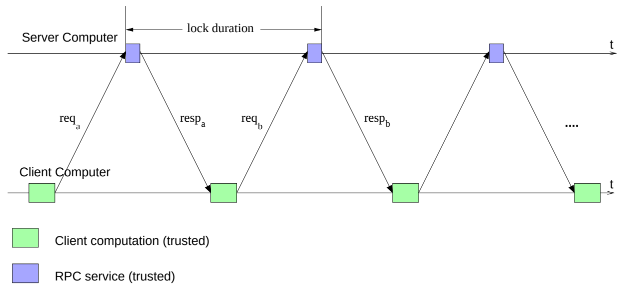
The S^{2}PRE/McRPC approach does not completely mask latency-induced delays. One situation where remote execution is desirable is when multiple clients access shared resources maintained by the McRPC server. In such a situation, proper locking is required for concurrency control on operatons on shared objects. Figure 4 shows the duration that object locks must be held for both the RPC and McRPC case.

3 Performance

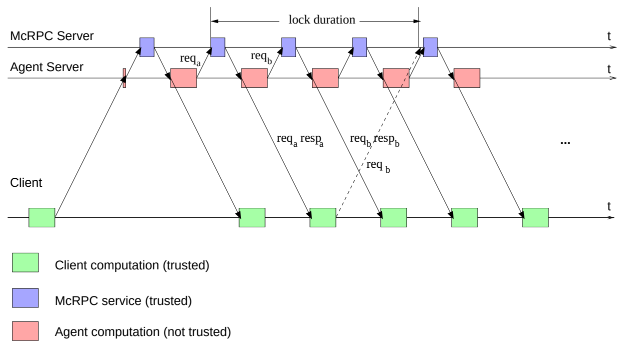
The pictures look good, but when, if ever, will McRPCs actually improve performance?

Suppose we have a (hypothetical, sustained) 100 Mbps long-haul network, with a 100 MB program image. The 800 Mb image transferred at 100 Mbps requires 8 S of transfer time, with one round-trip delay (RTD). If the application needs to perform k RPCs, the McRPC code would cost $8\text{ S} + \text{RTD}$, and the standard RPC version would cost $k \cdot \text{RTD}$. For RPCs from the western United States to western Europe, RTD is approximately 150 mS, so McRPC is appropriate when

$$8\text{ S} + 150\text{ mS} < k \cdot 150\text{ mS}$$



(a) RPC locking



(b) S^2PRE locking

Figure 4: McRPC Locking vs RPC Locking
Comparing improved locking with McRPC and RPC, we see that the S^2PRE-induced locking overhead can be small.

or $k > 54.3$.

4 Conclusion / Future Work

While most distributed computation do not currently operate in a latency-dominated environment, McRPCs appear to be attractive when many communication roundtrips are needed, allowing system designers to trade remote *untrusted* CPU cycles for improved time-to-completion. An important consideration, of course, is whether the security model makes sense: will it always be the case that *trusted* CPU cycles are available at or near the remote RPC server?

The McRPC model appears to benefit from technology trends. As CPU cycles and bandwidth become cheaper, trading off remote cycles for faster jobs will make more sense. However, as systems become faster, the size of problems that we wish to solve will concomitantly grow. With larger problems, the CPU demands and memory footprint will increase, and the portion of execution time due to communications latency may yet remain small. The communications delays at macroscopic distances also cause problems for internal signal paths within the CPU, and there is the additional possibility that increased CPU size and complexity—and decreased feature sizes—may pose a limit to the technology trends that have fueled system performance improvements for the last three decades. If such limitations arise sooner than expected, the McRPC model may never accrue enough benefit from technology trends to be truly compelling.

Experimental validation of the McRPC model and measurement of actual overheads involved would give a clearer picture of when McRPCs would actually improve performance for distributed systems.

References

[1] Christian Cachin, Jan Camenisch, Joe Kilian, and Joy Müller. One-round secure computation and secure autonomous mobile agents. In Ugo Montanari, Jos P. Rolim, and Emo Welzl, editors, *Proceedings of the 27th International Colloquium on Automata, Languages and Programming*, volume 1853 of *Lecture Notes in Computer Science*, pages 512–523. Springer-Verlag, 2000.

[2] Fay W. Chang and Garth A. Gibson. Automatic I/O hint generation through speculative execution. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pages 1–15, New Orleans, Feb 1999. USENIX.

[3] Chi-Keung Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 40–51, Goteborg, Sweden, June 2001. ACM.