Lila: A Cheating-Resistant Distributed Game Engine

Leo Martel, Susan Tu, Andy Moreland
{lmartel, susanctu, andymo}@stanford.edu

December 13, 2014

1 Introduction

In this paper we present Lila, a distributed multiplayer game engine that is Byzantine fault-tolerant. Lila allows games to be written with little regard for issues such as move replication, rejection of cheating moves, allowing new players to join the game, allowing players to leave the game, and removing cheating or network-partitioned players from the game. Lila is based on *Practical Byzantine Fault Tolerance* [1] and we seek to demonstrate through benchmarks that Lila is a practical underpinning for multiplayer games.

1.1 Background and Motivation

In multiplayer games, players are adversarial but must reach consensus on the state of the game. This consensus is achieved by designating a "host" server that receives inputs from all players and informs them of the results, which are taken as absolute truth. Sometimes the host machine is a coordinating server owned by the game company called a dedicated server, and sometimes players with extra computing power and (hopefully) stable network connections volunteer to host game instances. However, both of these approaches have drawbacks. Dedicated servers can incur large costs for publishers of popular games, and player-hosts must be trusted not to cheat because they unilaterally evaluate the state of the game. Lila presents a third option: the Byzantine-fault tolerant game engine detects and prevents a player-host's cheating, and enables a new player to take over as host without losing the partially-completed game.

In addition, multiplayer games have several unusual requirements that set them apart from other networked applications. Relatively little data is transmitted (just player commands and small state deltas), but low latency is paramount. Games are not only permitted but *encouraged* to drop slow clients; one laggy player can ruin the experience for everyone else. One common performance optimization is client-side prediction [7]: rather than blocking until the host reports the results of a game move, clients assume that their move is permissible and apply it immediately. Upon hearing back from the host, clients roll back rejected moves–this creates a startling "jump" for the player, but these rollbacks happen rarely enough for honest clients that client-side prediction is justified by the speed boost it provides.

We designed Lila to perform well under these constraints. View changes can be triggered on timeouts to ensure host latency never interferes with the game. Although a Byzantine fault tolerant protocol will always result in higher latencies than a trusted-host protocol, our benchmarks indicate that the latency penalty incurred is often acceptable. In addition, Lila supports predictive optimizations: by immediately applying local moves and applying remote moves on *PRE-PREPARE*, we expect Lila to achieve latency comparable to the standard trusted-host performance with client-side prediction. Running the distributed fault tolerance protocol on each move asynchronously still protects the sys-

tem from cheaters, with the caveat that cheaters are caught after a slight delay instead of immediately. This is an acceptable tradeoff—current game engines use manual administrator action to detect and ban cheaters, so optimistic Lila still reduces the potential cheating window from minutes, hours, or days to milliseconds.

2 Design Overview

2.1 Protocol for Byzantine Threats

Our protocol is heavily based on *Practical Byzantine Fault Tolerance* (PBFT). In our setup, one player is simultaneously a client and the primary (also referred to as the server). All other players are simultaneously clients and replicas. We assume that the players the game starts with can obtain each others' IP addresses, thrift ports, and public keys from some trusted 3rd party. (In the description below we assume $3f + 1$ players, but our implementation correctly handles any number n of players so that we can tolerate up to $\lfloor \frac{n-1}{3} \rfloor$ cheaters.)

2.1.1 Submitting and Approving a Move

Suppose we have some client A who wants to play a move. Client A sends this signed move to the server. (This signed move is the same as the client's request in [1]): $\langle$*MOVE, move_info, timestamp,*A_{ID}*, signature* $\rangle$. This move is run through PBFT. If the move looks valid, clients should eventually multicast a *COMMIT* message indicating this. If the move is not valid, then we continue with the commit process but refuse to apply the operation to our game state. (Note that we can skip the last reply to the client in PBFT since our client is the same as one of the replicas, and PBFT guarantees that once a move commits at an honest replica, it will eventually commit on at least $f+1$ honest replicas.) A replica that detects a cheater sends a $\langle$ *MOVE, kick out A, timestamp, replicaID, signature*$\rangle$ as a request to the server. To prevent many replicas from submitting a request to boot A at the same time, a replica should wait a randomized timeout before submitting this request. We consider it not particularly important to boot cheating clients the instant they cheat, so the lag that results from this timeout is acceptable in non-turn-based games.

2.1.2 Player Join

New player P sends public key to all players and then sends a $\langle$ *MOVE, join, ...*$\rangle$ to the server, which is run through PBFT. Non-faulty clients should all approve this unless the game is full.

2.1.3 Player Leave

If the player P intends to leave, P should send a $\langle$ *MOVE, leave,...*$\rangle$ request to the server, which is run through PBFT. Non-faulty clients should all approve this.

If the server S claims to have not heard from P in a while, it may be that (1) P has disconnected and should not longer be considered to be a part of the game or (2) the server is malicious. The server will run the request $\langle$ *MOVE, kick out P, timestamp,* S_{ID}*, signature*$\rangle$ through PBFT. In case (1), non-faulty clients will also not have heard from P, so they will vote for Ps removal. (Note that while it is possible that network issues have partitioned P from the server but still allow it to contact other clients, thereby preventing the server from kicking out P, we dont expect this to be the common case). In case (2), clients should vote against removal and remember that they refused a removal request from this server. Clients who voted against the removal should initiate a view change. After the view change, some client that voted against removal of P should issue a $\langle$ *MOVE, kick out S, ...*$\rangle$ request (clients should do this after a randomized timeout so that there are not many of these all at once). All honest clients that voted against removal of P are expected to vote for this request.

For the purposes of voting against dishonest attempts to remove players who are still responsive, the clients/server should keep track of who they

got a *PREPARE* from even if they no longer need that *PREPARE* message to consider themselves prepared. The same should be done with with *COMMIT* messages.

2.1.4 Malicious Server

Due to moves needing to be signed by the players proposing them, a malicious server cannot spoof moves; it can only fail to send *PRE-PREPARES* for them or be very slow to send *PRE-PREPARES* for them. Clients can time how long it takes for moves to be considered committed-local and initiate view changes if the average time is considered to be too long. They can also periodically request that they receive a notification from all clients when some request has committed so they can time how long it takes for the move to be committed everywhere, taking the average time of the $2f + 1$ fastest to ensure that malicious manipulation is avoided.

3 Implementation

We implemented the core logic of our distributed log and game engine in Java, using Apache Thrift [5] for Remote Procedure Calls to coordinate log replication. The game client communicates (locally) with the game engine using Thrift as well, so the client can be created in any language with minimal knowledge of the fault tolerance protocol or coupling with our game engine implementation. We are building a demo Chinese Checkers client in JavaScript. See Figure 1 for a diagram of our architecture.

For message signatures, we use the standard library (java.security) implementation of the Digital Signature Algorithm (DSA) with a 1024-bit key length. We intend to revisit this choice of signature. Our benchmarks (See Section 4) have shown that the bulk of our system's runtime is spent computing and verifying signatures. Not only is DSA slower for verification than equivalently-secure algorithms like RSA [6], but DSA provides much more security than is likely needed in practice. Multiplayer video games are in most cases both low-stakes and transient; a signature scheme that buckles after an hour of brute force attacks would still be sufficient to secure an entire Counter-Strike match. We plan to experiment with different choices of signatures and present several appropriate options to users of our library, allowing them to configure the tradeoff between signature speed and strength as needed.

4 Benchmarking

4.1 Experimental Design

We are interested primarily in the latency of move commits for clients so we designed a test to measure the average move-to-local-commit latency of moves being made in a round-robin fashion (simulating a game of chinese checkers) by a varying number of clients. Because we are interested in the common case of successful operation we do not include any view change events or other complications; we evaluated view changes in a separate benchmark.

Initially we attempted to execute our benchmark by distributing replicas across Corn machines, but our latency numbers are low enough in some cases that the results were significantly disturbed by the variable CPU load imposed by other clients on the machines. In order to compensate for this we executed the benchmark locally on a machine with 16 gigabytes of ram and a 3.8ghz quad core i7 processor. We believe that this is a reasonable decision because with optimizations discussed in Section 6 we expect that our latency overhead will primarily be due to signature computation rather than network communication. Therefore, it makes sense to benchmark in a situation that does not incur communication overhead.

4.2 Results

We executed our benchmark for replica counts ranging from 6 to 16. This is a significant slice

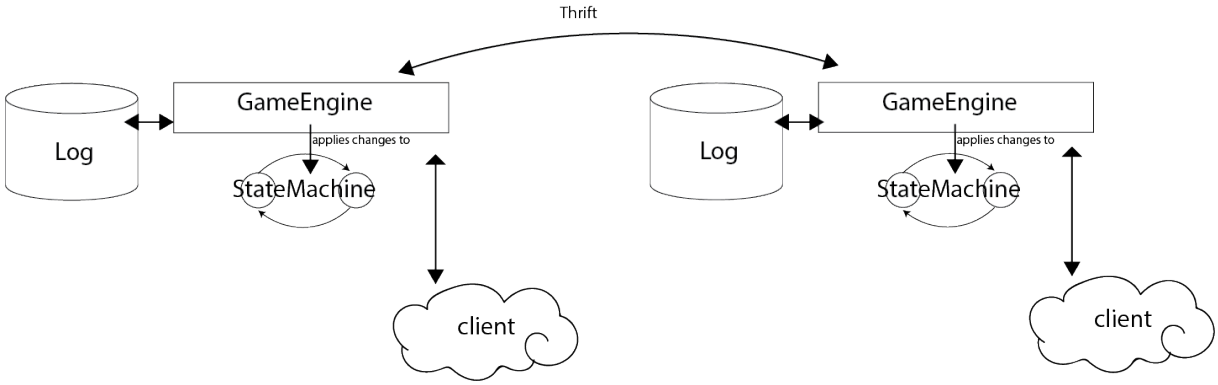

Figure 1: Two replicas communicating over Thrift. The game engine applies moves to the state machine after they are committed in the log. It sends the client's moves to the primary, which initiates the protocol by issuing $PRE-PREPAREs$.

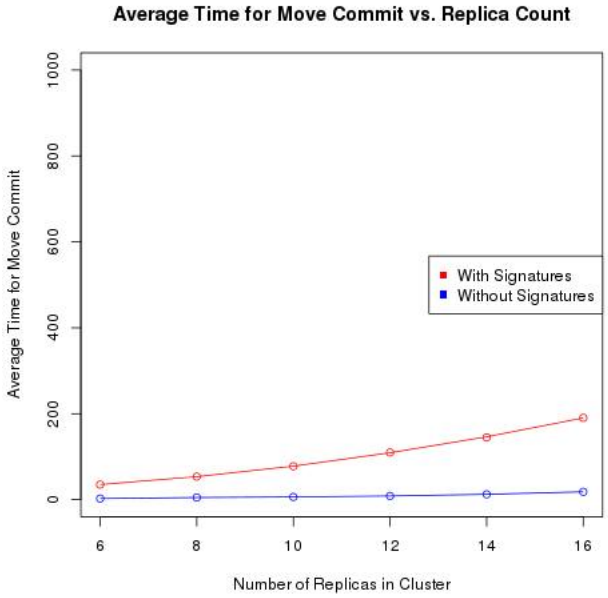

Figure 2: The growth of the average latency for move commits versus the number of replicas in our cluster. We see that the growth is roughly quadratic, and that the implementation that skips the signature computations is much faster in general.

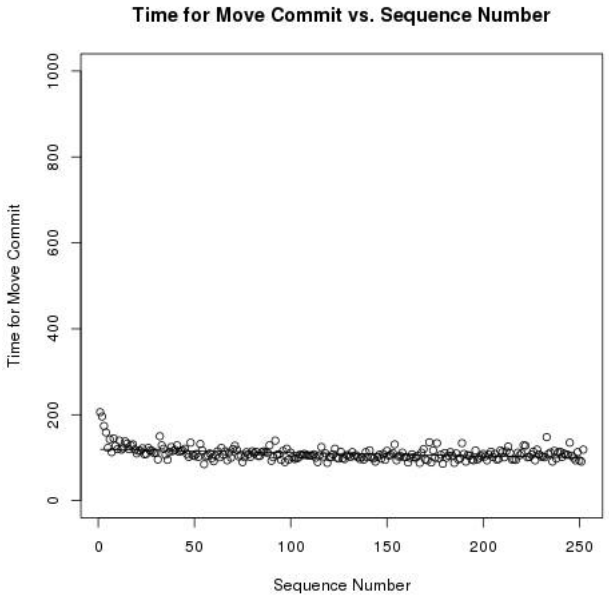

Figure 3: The latency of move commits as the sequence number increases with 12 replicas. We see that initially the latency is highly variable, but that this variance decreases with time. We believe the variance is due to a cold JVM.

of the range of player counts that we expect in most games that would use our library.

Figure 2 demonstrates a few notable things about the latency of our system. As expected, we see that the growth in average move-commit latency scales quadratically with the number of replicas. This is reasonable because the benchmark is running against an unoptimized implementation of our protocol which does not optimistically apply moves before commits happen. Therefore, we require roughly $O(n^2)$ messages to be sent during the prepare and commit phases of the protocol which accounts for the observed quadratic growth.

Additionally, we see that there is a large gap in la-

tency between the standard DSA implementation and an implementation that does not do any message signing at all. This justifies our assumption that signatures constitute a significant portion of our latency overhead and strongly suggests that a more reasonable signature scheme would yield large improvements in latency.

Figure 3 demonstrates that commit latency is relatively stable over time. Initially we see a mild drop in latency, but we believe that this is accounted for by the JVM's warm-up time. We see that as the sequence numbers approach roughly 30, the outliers start to drop out and performance stabilizes. This is the behavior that we expect because all of our log and state machine operations are more-or-less constant time, and checkpointing ensures that our log does not grow large enough for memory pressure to be an issue.

Figure 4, which shows results from separate benchmark of the average time it takes for replicas to progress from initiating view change to entering the new view, shows that view change latency is highly variable, with view changes done when we have to reiterate 9 operations (so that the primary can ensure that there is one operation per prepared sequence number in the new view) since the last checkpoint taking 30 times as long as view changes that occur immediately after checkpoints. These long view change times are probably a result of the size of the messages that need to be signed, i.e., when the new primary signs a *NEW VIEW* message, it is signing a message that contains $2f+1$ messages used to initiate the new view; each of these messages contains pre-prepares and each pre-prepare is accompanied by $2f + 1$ prepares. With 7 replicas, the *NEW VIEW* message for the view change that occurs 9 operations after the last checkpoint contains 225 prepares.

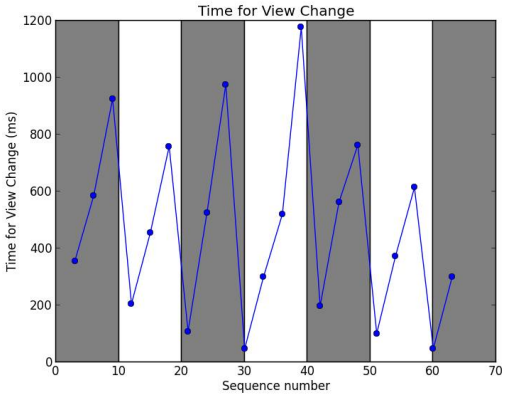

Figure 4: Each vertical stripe represents a period of time between checkpoints, i.e., we are checkpointing every 10 operations. Each point in the graph represents a view change. The benchmark view changes every 3 operations. There are 6 replicas here.

5 Related Work

We considered several other strategies for attaining correctness in the presence of Byzantine nodes. SUNDR [2] is a network filesystem that allows clients to verify data they obtain from an untrusted server by comparing the history that the server presents to them with the histories that the server presents to other clients. As in PBFT, signing makes it impossible for the server to spoof operations and insert them into the file history. It must also always present a client with that client's own operations; therefore a malicious server can only make a client miss operations from other clients, a type of attack that would be detected if clients compare histories. However, we chose not to base our implementation on SUNDR because while SUNDR's advantage over BFS is to make attacks detectable without replication, we do in fact need replication of the game state at each client.

In terms of other Byzantine fault-tolerant consensus algorithms that we could have used to vote on the validity of moves and remove malicious servers, Lamport describes an extension of multi-paxos in [3] that handles Byzantine failed nodes. [4] notes that it may be difficult to distin-

guish malicious behavior from transient network problems in PBFT (perhaps leading to more view changes than necessary), but in our case, since we want to boot players who have poor connections, we have little need to distinguish.

6 Future Work

[1] suggests that replicas execute a request tentatively as soon as the request is prepared. Since the common case in our use case is that moves will be valid and should be committed, this would be an appropriate optimization to attempt; we suggest being more aggressive and applying as soon as it is pre-prepared. Our current implementation's latency overhead compared to what would be necessary with a standard game engine (which just needs to send the client's move and receive the result) is an an additional 2 rounds of messages; if we allow moves to committed as soon as they are pre-prepared, then we have the same number of rounds as a standard engine. We expect this to reduce our overhead from a multiplicative factor of around 3 from the stack of RPC calls to a reasonable additive factor incurred by signature computation.

View changes that occur well after checkpoints are currently very slow and could probably be improved by signing message digests or by switching to MACs.

Player joins/leaves are not yet supported.

7 Conclusion

Our benchmarks show that Lila's latency is certainly low enough for turn based-games. With additional optimizations (such as application of pre-prepared moves, using MACs as suggested in [1] instead of digital signatures, which are expensive, especially when the game has many players whose signatures we must verify), and client-side prediction, it may be low enough for real-time games as well.

Our source code is available at `https://github.com/AndyMoreland/cs-244-project`.

References

[1] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. *Proceedings of the Third Symposium on Operating System Design and Implementation*, 1999.

[2] J. Li, M. Krohn, D. Mazieres, and D. Shasha. Secure Untrusted Data Repository (SUNDR). *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, 2004.

[3] L. Lamport. Byzantizing Paxos by Refinement. `http://research.microsoft.com/en-us/um/people/lamport/tla/byzsimple.pdf`

[4] L. Lamport. Leaderless Byzantine Paxos. `http://research.microsoft.com/en-us/um/people/lamport/pubs/disc-leaderless-web.pdf`

[5] Apache Foundation. Thrift 0.9.2. `https://thrift.apache.org/`

[6] `http://msdn.microsoft.com/en-us/library/ms978415.aspx`

[7] Y. Bernier. Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization. `http://www.vis.uni-stuttgart.de/plain/seminare/computerspiele/latency.pdf`