Viewstamped Replication
Visualized

SCOTT MENDOZA, JORGE OCHOA
Stanford University
December 13, 2017

Introduction

Distributed algorithms are inherently difficult to reason about in part due to the chaos of interactions between multiple agents and the uncertainties in the network. The papers written for these algorithms increase the difficulty by often struggling to balance sufficient detail with conciseness. At times, a paper may elaborate too much on one aspect, confusing a reader's understanding of the big idea, or it may remain too vague, leaving ambiguities for the reader to interpret.

Visualizers that simulate the algorithm can provide a complementary learning tool to readers, presenting the big picture of an algorithm at a glance, while still revealing crucial details by encouraging users to interact and experiment with the protocol. As such, we have created a visualizer that simulates Viewstamped Replication (VR). We were inspired by the Raftscope visualizer's effectiveness in depicting Raft. To us, the Viewstamped Replication paper often felt too vague and confusing to understand, and we believe that a visualization could quickly capture the essence of the protocol.

We will first provide an overview of the visualizer and its features in Section 1. Then, in Section 2, we will discuss our implementation of the visualizer, and in Section 3, we will talk about our design decisions. Finally, in Section 4, we will talk about the challenges that we faced.

1. Overview and Features

Our VR visualizer is implemented as a webpage, using primarily Javascript and HTML. By default, we have five server nodes (this number is configurable) composing a server group and a client node which issues transac-

tions to that server group. Users can see messages travel between nodes, and each message is represented by a keyword indicating its type (e.g. ♡ for heartbeat, INVITE for a view invitation). For each node, we show key information, such node ID, current viewid, and log as seen in Figure 1.

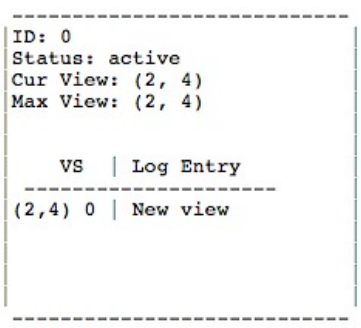

Figure 1: Server Node

The information shown in the table changes to show election information whenever the node is involved in an election, and the user has the option of viewing the heartbeat timeouts instead of the log. We allow the user to pause or resume the simulation, start or commit a transaction, and kill or revive any of the servers.

2. Implementation

The simulation runs in iterations that occur about every second. We decompose the algorithm into 2 types of entities, namely server and client. During each iteration, all nodes perform their respective duties, such as receiving and sending heartbeats, processing operations associated with transactions, or participating in view-changes.

We simulate messages being transmitted through a network by holding them in a central queue for a semi-random amount of time and delivering them to the destination once this time has elapsed. We do not allow for messages to be explicitly dropped in transit, although if a server is crashed, any messages to it will be dropped upon arrival.

While there is the notion of data (gstate) that is being replicated in the VR paper, we do not actually simulate any real data objects. Instead, we simply record operations associated with transactions in a log that is replicated across nodes. If we were to actually simulate the data objects, the server replicas would actually apply operations to the objects and not just record them in a log.

Servers know the configuration (all nodes in the simulation) from the beginning. This differs from the notion of the current view, which is the set of active, participating nodes.

We will now go through a couple of the more difficult aspects of the VR algorithm and how we implemented and interpreted them.

2.1 View Changes

Our implementation of the view-change algorithm was mostly faithful to its presentation in the paper. Each server object maintains state tracking the last

heartbeat message received from a node. A server uses this state, along with whether it has just recovered from a crash, to trigger view-changes. The other server nodes enter the view-change underling state upon receiving an invitation (assuming it is valid to accept). We follow the paper's rules governing how to determine a successful view election, which involves taking into account "crashed" and "normal" acceptances. We follow the paper to determine who becomes the primary of the new view and send an INITVIEW message to the new primary if it is not the view-manager.

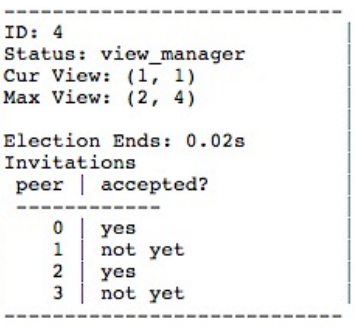

Figure 2: View Manager

Where we differ from the paper is in our implementation of NEWVIEW messages, which signal the beginning of a view. In the visualizer, the primary of the new view appends its viewid, history, and log in the message, and the backup servers in the new view copy this state entirely, ensuring that they all begin in identical states. The VR paper seems to imply that the entire gstate is passed around, so we emulate this by passing the history and log around. This would probably be too costly in a real implementation, but in our simulation, with limited state, this is fine.

2.2 Transactions

Our version of transactions is simple compared to the VR paper's. We implemented single-operation transactions and only allow one transaction at a time by restricting the client from beginning a secondary transaction until it commits or aborts its first. On the server's side, the VR algorithm allows a primary to lazily replicate transaction operations, at least until a client decides to commit the transaction. Instead, our simulation simultaneously sends the replica data to the backups. Backups take the event record for an operation, update the viewstamp in their history, and append both to their log.

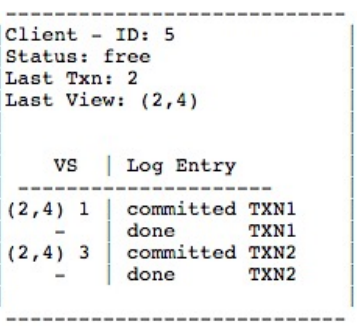

Figure 3: Client Node

We did not implement a communication buffer that guarantees in-order delivery of messages. Instead, because we know that messages are never dropped, we know that a replica will either eventually receive a message or has crashed. This means that when the client decides to commit a transaction, we can just wait for confirmation that all backups have received a transaction

operation or for a view-change to be triggered. To check whether or not a client has received a given operation, we have the primary peek into each replica's state (possible because all servers are global objects) to see if the operation has arrived.

3. Design Decisions

Because the goal of the visualization is foremost to be a learning tool, we made certain design decisions that either allowed us to simplify our code or more clearly present the algorithm. For example, our visualization only simulates one server group, whereas the VR protocol allows for many server groups. Although we can create more servers and server groups are mostly self-contained, adding server groups would increase the complexity. In its current state, the client assumes that there is only one server group. This enabled a more simple implementation of the client because the success and failure modes of a transaction are only dependent on a single server group. Along the same lines, we only implemented single-operation transactions, although we feel that this is sufficient to get the idea of the protocol across.

In the VR protocol, the client is replicated, whereas in our simulation, we have an unreplicated client. This was done both to simplify our code and to clearly differentiate a client from a server in the visualization. When first reading the VR paper, a major point of confusion for both of us was the fact that there are multiple primaries, because all parties involved in a transaction are replicated. Having a single client node provides a point of centralization that helps the user follow what is going on. For simplicity, we do not allow the client to be crashed.

As mentioned in Section 2, we do not actually implement data-objects or files in server-groups, and instead, we maintain a replicated log. We feel that showing the operations in the log is sufficient to see the replication provided by server groups as part of the protocol. We did this because the goal of the visualization is to help users understand the VR protocol, and not to actually implement a potential application of it (i.e. a database or file system).

The VR protocol relies heavily on the underlying communication buffer in each server, which guarantees in-order delivery of messages. We structured our implementation to not require a buffer, as this would require us to implement something akin to TCP, keeping track of the last message acknowledged by each server. This ties into the decision to not allow for dropped messages. By not allowing for dropped messages, we can also more easily reason about the state of a message because it is either delivered or the destination node has crashed.

4. Challenges

We found it difficult to balance being faithful to the VR algorithm with taking advantage of our simulation environment. We wanted to accurately depict VR, but in many places, it leads to complicated code that the user does not benefit from. For example, we implemented heartbeats by making servers expect periodic heartbeat messages from each other. However, in our code, the server objects have access to any node's state, so we could have just directly checked the state of any other server.

In other places, we took advantage of the simulation. For instance, the client always knows who the primary of the current view is because we peek through each server's state. This simplified code in an area of the visualizer that matters little to the user. We take advantage of timing and a packet's inability to be dropped to know when a transaction is ready to commit. In an actual VR implementation, this is impossible, but this goes hand in hand with making a learning tool and not a practical implementation of VR.

Developing our simulation in Javascript also lead to challenges by introducing bugs that could have been caught by the compiler of a stricter language. It becomes much easier to make mistakes when you can accidentally add member variables to objects instead of accessing existing ones. Buggy code that in another language would not even compile runs without so much as a warning in the browser. This, combined with the distributed nature of the algorithm, sometimes made it very difficult to debug.

We faced challenges deciding on how much and what to display to the user. As we display more to the user, we potentially clutter our visualizer and decrease its effectiveness a learning tool. For instance, we feel that a server group of five nodes illustrates the point of replication better than a three server cluster. Howevever, this takes up more space, especially if we had decided to introduce multiple server groups in the simulation. We chose to show a limited display of the log, but we could have also chosen to display the entire log.

Translating the VR algorithm from paper to Javascript was our greatest challenge. The original VR paper can feel ambiguous at certain points. Often, the key detail needed for our implementation was a single sentence hidden in an obscure paragraph or not explained at all. The relationship of timings in the VR protocol (like heartbeat timeout or election timeout) is also not directly stated, so we simply constructed arbitrary values that seemed to work well. We found the naming of state variables, like the server's history, misleading. Given the name, we assumed that the history was a sequence of viewstamps, when in fact, it just maintained the highest timestamp for each view. We only realized this once we were implementing the compatible check for committing a transaction. We also found the paper to be obscure

regarding how a backup server processes an event record. Because we had no gstate, this was not really a problem, so we simply added the event record to our log.

Conclusion

We created a visualization that simulates a subset of the VR protocol. Our visualization had to cut some corners, but we tried to only do this where it would not significantly detract from the big ideas of the protocol. We have implemented the view-change algorithm, which is the meat of the paper, and most other features. The scope of our simulation and our implementation of transactions stand to benefit the most from improvements. For future work, we would implement multiple server groups and allow multi-operation transactions involving more than one server group. Despite the simplifications that we made, we believe that our visualizer is still an effective representation and learning tool for the VR protocol.