Mukundan Kuthalam, Shantanu Kanvinde, Paul Hainey, Matthew James

UTEIDs: mk33274, smk2842, prh488, maj2763

Code repository: https://github.com/smkanvinde/EE360P-Project

Testing Distributed Systems

Abstract

The most difficult part of debugging distributed systems is not having any control over message communication. In order to give the user such control, the team built the Visual Dashboard. Acting as a coordinator for message passing, the Dashboard intercepts each message before it reaches its target, allowing users to change the system state as they please. The user can change the order of messages, change the message itself, or delete the message. The algorithm itself works well, but has a drawback in its use of a centralized algorithm.

Intro

Software testing is generally a difficult endeavor, but testing distributed systems is especially complex due to the nature of messaging and the problem of capturing a global state. The asynchronous distributed system model does not guarantee an upper bound on message delays. This means that messages may take a long time or even get lost forever. Further complications arise when the problem of concurrency is addressed in relation to actually processing and acting on these messages that are received. Mutual exclusion must be satisfied as each distributed node requests shared resources in order to guarantee an accurate result. This document will further explore the difficulties of testing distributed systems along with the motivation behind our project, as well as how our Visual Dashboard design can be used to help debug such programs and the limitations that would need to be addressed by any system that seeks to accomplish more.

Motivation

What makes testing distributed systems hard is the system's reliance on messages, which are the one thing that users cannot control to a desirable extent. Many algorithms have been developed and proven to work for solving common distributed system problems, but these theories need to be coded and tested. Traditional debuggers just step through code without giving users a way to observe several threads at once. Opening multiple Java Virtual Machines and running each thread as a separate machine is inconvenient and says nothing about messages in-transit. For example, it is possible that three messages are being sent from process A to B and one of them has been lost, while of the remaining two messages, the one that was sent first arrives last.

So, we can conclude that a necessary part of debugging a distributed system is allowing users to simulate message delivery, so that they can ensure their system works for even the strangest delivery and receive events. In order to achieve this, the Visual Dashboard allows the user to see the state of the entire system upon a message being delivered or received and have the user modify the message list in various ways.

Visual Dashboard

With the Visual Dashboard, the user can adjust the timing of the delivery of messages and even whether these messages are delivered at all. The Visual Dashboard is a queue of all the messages sent by every process and allows the user to adjust the order in which messages are delivered and also delete these messages.

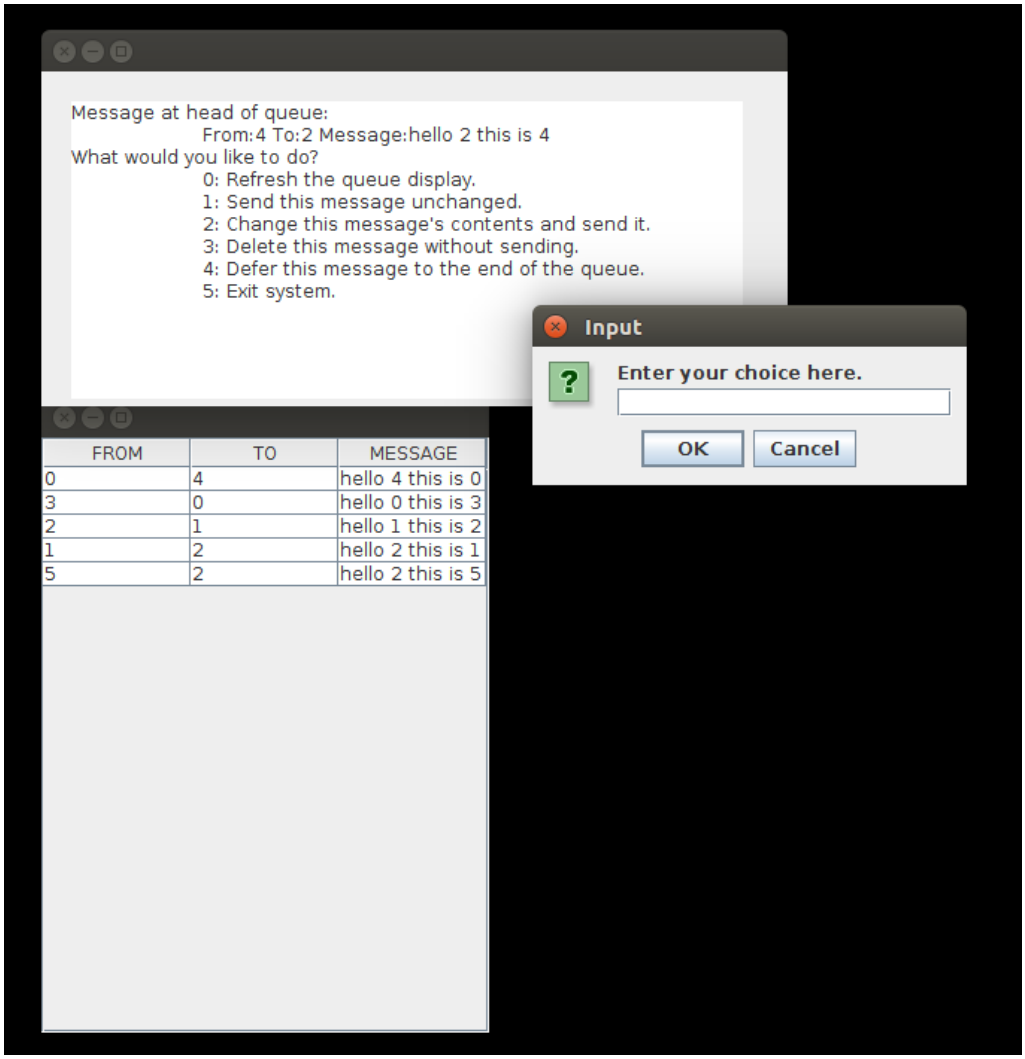
Often times, we have processes that want to send messages to every other process, and this means it can be hard to notice when and which messages have been sent at any given time. The easiest way to detect all the messages in a system is to have them go through a coordinator process that manages everything message related. Thus, the Visual Dashboard does just that. With the coordinator, which is a pipeline for all traveling messages, we can, with some certainty, ensure that the user can see all the messages that are in transit across the

system. Of course, this still has a limitation when it comes to the time it takes for the messages to travel. It could take anywhere from a few seconds to an hour for a message to travel between two processes.

In order to give the user time to adjust the system in a timely manner, each client currently waits 30 seconds between sending messages. This way, the user actually has time to see what just happened and modify the queue to match whatever testing scenario they would like. Using some basic information about the messages, the user can delete messages and the order of message delivery can be adjusted, thus ensuring that there are no synchronization or reliability issues for the code under test.

In this way, the Visual Dashboard is a central coordinator that simply contains a queue of all messages and these messages can then be adjusted in such a way that the user can tell how their code behaves under a certain scenario.

Results



Shown above is a screenshot of our system on startup. We can see that there are six processes running - one whose message has been de-queued and is shown in the top panel, and five whose messages are still in the queue. For human workability, each Client only sends one message every 30 seconds. We can refresh the queue and see the new messages using option 0. Option 1 allows us to send the message at the queue's head to its intended recipient without any changes. Option 2 lets us change the contents of the message itself. Option 3 deletes the message from the queue without sending it to its recipient, and option 4 changes the message receipt order by deferring the message at the head of the queue to the tail of the

queue. Option 5 exits the system by simply calling System.exit(0). Each module was tested individually. One limitation of the current implementation is our use of javax.swing for the GUI. We currently close and reopen each panel on each action to refresh. We are also limited to the functionalities provided to us by the Swing API. A more robust GUI might be better suited to a program such as this. Ideally we would want one that not only updates the queue in real time, but we would need one that is able to process and update information fast enough to model a realistic system when we increase the message speed to faster than one message every 30 seconds. The full functionality of our system will be demonstrated in lecture.

Alternate Design Choices

The code works but has limitations. The limitations of this system are found in its centralized algorithm and the coordinator's system wait, which both affected our approach to the code design.

The centralized algorithm approach was taken because collecting all the messages in the system can be a rather complex task for obvious reasons. This brings up the usual issue: The coordinator crashes. After all, the coordinator is just another process, and reliance on any one process in a distributed system makes for a clear lack of fault tolerance. Thus, one alternative that was considered but ruled out was building a queue that could jump across processes to collect messages, but then it is hard to time when the queue should jump to another process so that it gets the message.

Thus, as we can see, the Dashboard suffers from its centralized algorithm, and that can lead to certain issues when testing distributed code.

Conclusion

As one can see, the most difficult part of debugging a distributed system is understanding how the code can work when faced with the various nuances of the “process message delivery system.” Thus, the Visual Dashboard offers a centralized approach to giving the user a way to modify the message delivery system, so they can see what goes wrong and why in various scenarios. However, the Dashboard is still another process and uses a centralized algorithm, thereby naturally suffering from limitations that would less likely be present in a more distributed debugger. The Visual Dashboard is certainly a step in the right direction of solving the problem of distributed programming, but as with all first steps, suffers from issues that we hope will be resolved in further refinements of this product.

Appendix

Educational Material: Testing Distributed Systems

The asynchronous and nondeterministic nature of real world distributed systems can cause many issues when attempting to test them. When a distributed system is asynchronous, messages between nodes may take an arbitrary amount of time to send. We can describe an asynchronous distributed system as nondeterministic in the context of testing since we can never reproduce consistent message ordering and clock states. Both of these properties make traditional software testing methods like regression testing and end-to-end testing much more complex. It is also difficult to simulate real world use cases for distributed systems that include thousands of nodes all interacting at once. Testing distributed systems is a topic that is still being researched at an academic level rather than regularly practiced in industry. This appendix will explore some of the concepts and methods that are currently being studied.

One way to test distributed systems is to create multiple 'layers' of testing that depend on the nature of the specific system under test. Each layer contains well-defined, traditional software testing methods that validate the system at whatever level of abstraction the layer is responsible for. For example, it is common to start with a basic layer of unit tests that are run on individual machines by individual developers. This layer can be enhanced with VM integration in order to better simulate a real world environment. Overall, the first layer is responsible for validating correct behavior on one single machine, but it avoids many of the difficulties of distributed testing that were mentioned before.

The next layer consists of system-level integration tests with more than one node. This layer may include a sandbox environment that can be used to run various experiments and functional tests. The details of testing in this stage are almost completely determined by the specific system under test. Basic, vital functionality may be checked here, but this layer is

mainly responsible for ensuring that each individual module of the system can run simultaneously without error...

Another issue with debugging distributed programs is that most methods require the production of a large volume of data to analyze the execution traces of a program. In order to help programmers better analyze this trace data, it can be extremely useful to develop a system that helps the programmer visualize behavior patterns from the data. One type of such a system involves a causality graph that illustrates all the interprocess communication events of a program. However, illustrating every event can still produce a large amount of information that the user must evaluate.

A visual pattern matching paradigm would offer the programmer the ability to specify normal and abnormal patterns in a program as another method of visualization. This would give the programmer a way to filter the data towards what would be relevant for locating a bug in the software. Visualizing a *waits-for* relationship is an example that allows the programmer to easily locate any potential deadlocks. The other main benefit of this example is that the testing data produced by this relationship is only proportional to the number of processes, not the number of events in the program. This greatly reduces the data produced from testing and offers a unique perspective for the programmer to use to debug a system.

While visualization can be useful for debugging distributed programs, some scholars believe that such visualization requires intimate knowledge of the relation among processors and processes. That is why visualization should be paired with an added analytical capability, instead of visualization alone, so that the programmer can better focus on the more relevant data in debugging.

Visualization:

https://pdfs.semanticscholar.org/db97/a77f05e975784b892f515f64e50878768c87.pdf

Layers:

http://www.drdobbs.com/testing/testing-complex-systems/240168390

More complicated:

http://colin-scott.github.io/blog/2016/03/04/technologies-for-testing-and-debugging-distributed-systems/