Foundations of Distributed Systems – Lab Guide

José Orlando Pereira
HASLab/DI/U. Minho

November 12, 2012

The following projects aim at illustrating fundamental distributed systems modeling concepts and key distributed problems, concepts, and algorithms. A secondary goal is to experiment with an event-driven programming paradigm, contrasting it to the more common threaded approach to managing concurrency. Therefore, trivial protocols are proposed regarding message format and content, as well as functionality that is provided by target application scenarios.

1 Trading

Consider a server providing a basic trading service, matching sell and buy offers by different clients. The server never holds stock. Instead, acknowledgments are returned only when a match is made. The protocol is as follows:

- The server listens on two different ports for seller and buyer clients.
- Each byte (any value) sent on an existing connection signals an offer to, depending on the target port, buy or sell one item.
- Each byte (any value) returned on an existing connection acknowledges that one item, depending on the target port, has been bought or sold.

Steps

1. Implement the server using blocking sockets (e.g. `java.net`) and threads. Test the server using `nc` interactively as a client.
2. Rewrite the server using non-blocking sockets and select (i.e. `java.nio`). Repeat tests.
3. Refactor the server making sure that the main loop is independent of the business logic by encapsulating it in an abstract handler interface.
4. Implement a concurrent client that quickly makes a very large number of offers, while slowly receiving replies. Repeat tests with threaded and event-driven versions of the server.
5. Add to both versions of the server an initial log-in step for clients: The server prompts and user name, then a password.

Questions

1. Identify external inputs to each version of the server and the actions executed. Compare both versions.
2. Model each version as a state machine. Is the set of behaviors the same for both?
3. What is the preferred concurrency approach for each version (with and without login)? Can you generalize?

Learning Outcomes Recall basic distributed programming with sockets and threads. Implement distributed programs in an event-driven fashion with select. Recognize the equivalence between threaded and event-driven code and with an abstract state machine. Select the best approach, threaded or event-driven, for different purposes. Identify and differentiate safety and liveness properties of distributed programs.

2 Ring

Consider a distributed mutual exclusion primitive (mutex) using a ring. In detail, a group of peers is organized in a ring by establishing connections among them. The peer that owns the token, may allow a local thread to enter the critical section. When the thread leaves the critical section, the token is sent to the next process in the ring. A single byte should be used to pass the token in the ring.

Steps

1. Implement the peer. Assume that there is at most one thread in each peer.
2. Reconsider server implementations, assuming that there are a very large number of threads in each peer.
3. Assume now that each peer in the ring is a lock server, that serves a number of local clients. It should interpret an incoming connection as a lock request; it should reply with a single byte when the lock is granted; and should interpret a disconnection as a lock release. Implement each peer with a single thread and select.

Questions

1. Identify external inputs and the actions executed.
2. Model the system as a state machine. What properties should be ensured?
3. Can you argue the correctness assuming a single thread?
4. Reconsider correctness arguments of liveness properties assuming multiple clients.

Learning Outcomes Formulate a specification as an abstract state machine. Argue the correctness according to both safety and liveness properties. Locate fairness challenges and employ mechanisms that address them.

3 Bank

Consider a distributed bank that works as follows:

- Each process hols a single account.
- A process issues a sequence of transfers from its account to other accounts.
- A process will also handle incoming remote transfers to its account.
- A mutex (e.g., synchronized block) is held both while issuing outgoing transfers as well as when handling incoming transfers.

Since this is likely to cause distributed deadlocks, a mechanism to detect them must be in place. We also want to know it at any given time, transfers exceeding some global maximum are in progress.

Steps

1. Implement the base application and verify that it leads to blocking.
2. Using a monitor process, determine if there is a deadlock in the system.
3. Using a monitor process, determine if ongoing transfers exceed maximum.

Questions

1. Explain why the synchronous solution is undesirable. Can you show the problem experimentally?
2. What is the best approach for detecting the deadlock? Why? What if deadlock detection was to be performed by all participants themselves?
3. Does your solution guarantee that, in case all transfers exceed the maximum, that this is discovered?
4. When your solution discovers that all transfers exceed the maximum, can you guarantee that this was true at some point in time?

Learning Outcomes Identify and differentiate stable and transient global predicates of distributed programs. Recognize and explain the challenge in computing global predicates in distribute systems. Define causality and logical time. Design solutions for global predicate evaluation in an asynchronous system model with logical clocks.