IEEE distributed systems ONLINE
Expert-authored articles and resources

Causal Errors in Distributed Systems

Nikki Lesley, Holly Hou, and Joydeep Mitra • University of Sydney, Australia

This article presents a project that analyzes the pattern of causal errors in distributed systems. It asks the question: how wrong could the system be if causal order is not enforced? Simulations of a stock-trading system show that the number of failed trades due to causal errors has minimal impact on the system.

Building software has always been recognized as a difficult problem, and it's especially difficult in distributed systems. Message delays, different processor speeds, service access clashes, and many other issues add to the complexity of the software development process.

In a message-passing distributed system, application developers must worry about message ordering. Some researchers have argued that this should be the application developer's concern[1, 2], while others argue that it should be left to the underlying communication system[3]. This debate is based on the premise that message order errors are bad and must be handled at either the communication or the application level.

We choose a third approach, posing the question: how bad are message order errors? In particular, we investigate causal ordering's effect on the correctness of distributed applications. We're interested in such questions as

How wrong would the system be if no restrictions on causal

www.manaraa.com

ordering existed?
What is the relationship between network delays and causal order violations?
Is there a relationship between a system's business logic and the percentage of causal order errors that cause problems?

We begin to answer these questions by providing motivation for this work and explaining the context in which we pursue it.

Motivation

For many years, developers have considered distributed systems as the "next big thing." The Internet, CORBA, Web services, and other related technologies have brought these systems popular acceptance. Much discussion as to the best way to approach distributed applications remains, and work to simplify building such applications is ongoing.

When building an application over middleware, developers can use remote method invocation semantics, which guarantee total order, or message-oriented semantics, which make no ordering guarantees. Because RMI semantics attempt to replicate traditional method invocation semantics, they reduce the parallel nature of distributed systems: Distributed pieces of the system must block until the remote method invocation returns. Message-oriented semantics maximize parallelism by continuing to process as they send out messages. At issue here is that messages can trigger other messages, possibly leading to a chain of causality that gets muddled without blocking.

Consider Figure 1. In Figure 1a, process C reacts to process A's second message before reacting to its first message. With RMI semantics, as in Figure 1b, this cannot happen because A won't send its second message until it receives the appropriate response to its first message. Similarly, with causal order guarantees, C will not see

المنارة للاستشارات

www.manaraa.com

A's second message until it has seen A's first message because the second causally depends on the first.

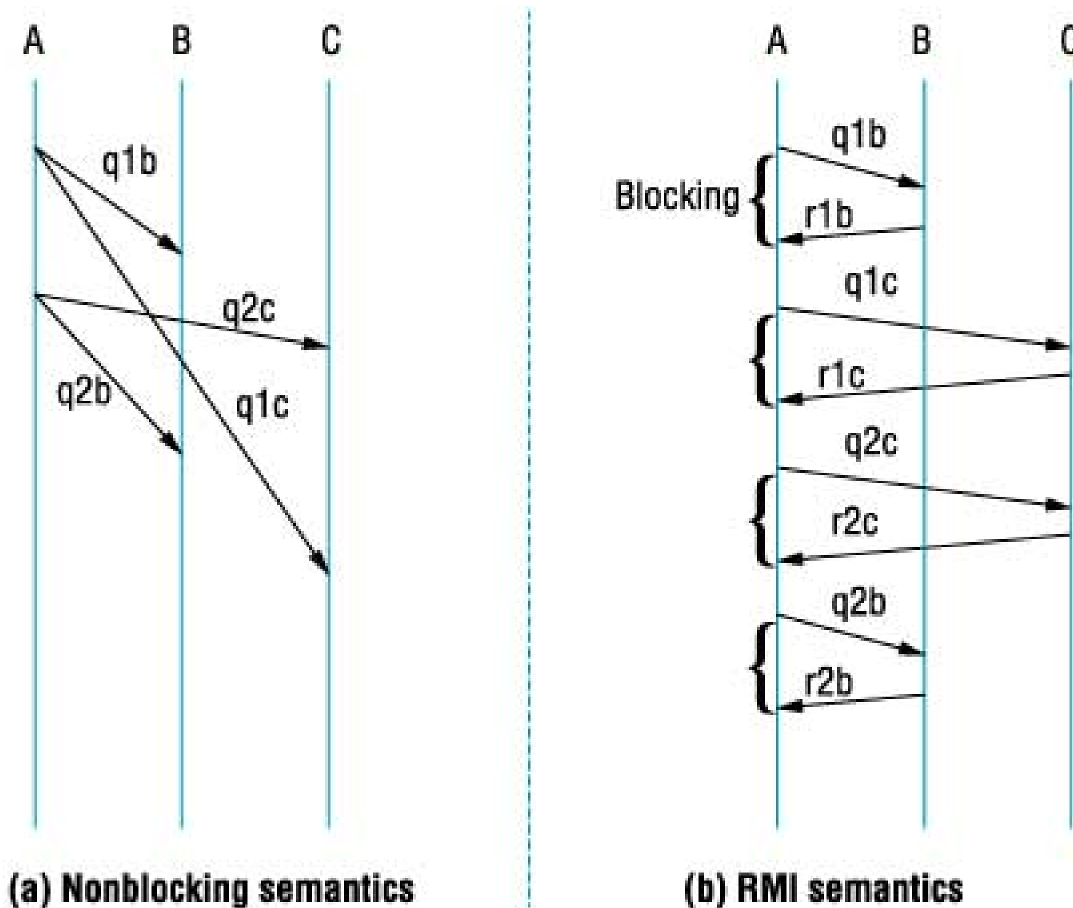

Figure 1. Message ordering semantics: (a) nonblocking semantics and (b) remote method invocation semantics.

Background

Two main schools of thought attempt to cope with message ordering:

- Leave all message-ordering decisions up to the application, the *end-to-end* argument[1,2]
- Make the communication subsystem enforce message ordering at an application-specified level[3-6], ranging from no ordering to total ordering

المنارة للاستشارات

www.manaraa.com

The end-to-end arguments raised by Jerome H. Saltzer, David P. Reed, and David D. Clark[2] suggest that functions should be implemented at the layer closest to where they are used. Lower layers shouldn't provide message ordering because they don't have enough information for the implementation, and because the applications that do not need the functionality might still have to pay for it. Although nontraditional network technologies such as active networks[7] and firewalls[8] challenge end-to-end arguments, it's a highly regarded communication protocol design principle.

On one hand, causal ordering is an application-level issue, so it seems natural to put it at the application layer. Doing so prevents efficiency loss in applications that don't need causal ordering. However, it is very difficult for application developers to be responsible for guaranteeing causal ordering in a distributed system for several reasons:

> Causal order is closely related to the business logic and is subtle and detailed; thus, identifying causal relationships requires adding large amounts of repetitive work to application development.
> Because the business logic can change from time to time, causal order will always be an open issue in the system.
> It is almost impossible to exhaust all possible causal error violation cases in large-scale application development involving large groups of processes.
> A goal of distributed system design is isolating application developers from lower-level complexity. Because causal ordering issues are related to asynchronous programming styles, having application developers handle the causal ordering can result in errors. The effort spent debugging might outweigh the benefit of not having the communication level handle the ordering. In addition, few adequate tools for dealing with the complex causality structure in distributed systems exist.

المنارة للاستشارات

www.manaraa.com

On the other hand, providing causal ordering at the communication layer violates end-to-end arguments. Business logic is the root cause of causal-ordered events. However, the communication layer can only see the sequences of the events, not the events themselves. As a result, causal relationships captured at the communication layer are only potential causal dependencies. Two events generated by the same source might or might not be causally related given the business logic, but the communication system will always consider them causally related. Hence, the system loses efficiency by being overly cautious.

Another major problem with providing causal ordering at the communication layer is the *hidden channel* issue. A hidden channel exists when processes share information in a non-message-passing manner, such as through a shared database. In this case, the communication layer will not see the causal relationship occurring independent of message ordering. Additionally, the cost of enforcing causality either through vector time stamps or other header information might be nonnegligible. Despite these drawbacks, communication-layer solutions appeal to many researchers[5,6,9].

Our approach tries to find the middle ground, whereby neither the application nor the group communication system provides message ordering. As long as ordering problems are not critical to the application and do not occur frequently, the ultimate goal is a system that notices when things go wrong because messages are out of order and tries to cope.

Implementation

To investigate the affect of causal ordering on distributed application correctness, we implemented a distributed application, removed the system's causal-ordering restrictions, and logged the results. We used vector time stamps[10] to monitor messages' causal relationships.

للاستشارات المنارة

www.manaraa.com

Security trading system

Our distributed application, MarketSim, simulates a distributed security trading system that lets different marketplaces trade with each other. Clients post their offers through brokers, and orders are broadcast to the marketplaces. After the marketplaces find a buy and sell match, the marketplace broadcasts the matching orders to update the databases at each marketplace.

Brokers communicate with each other directly to settle the match. Many business scenarios exist. However, all the possible scenarios are subsets of the business case shown in Figure 2, involving two brokers from different marketplaces. We added a third marketplace to illustrate the broadcasting process.

المنارة للاستشارات
www.manaraa.com

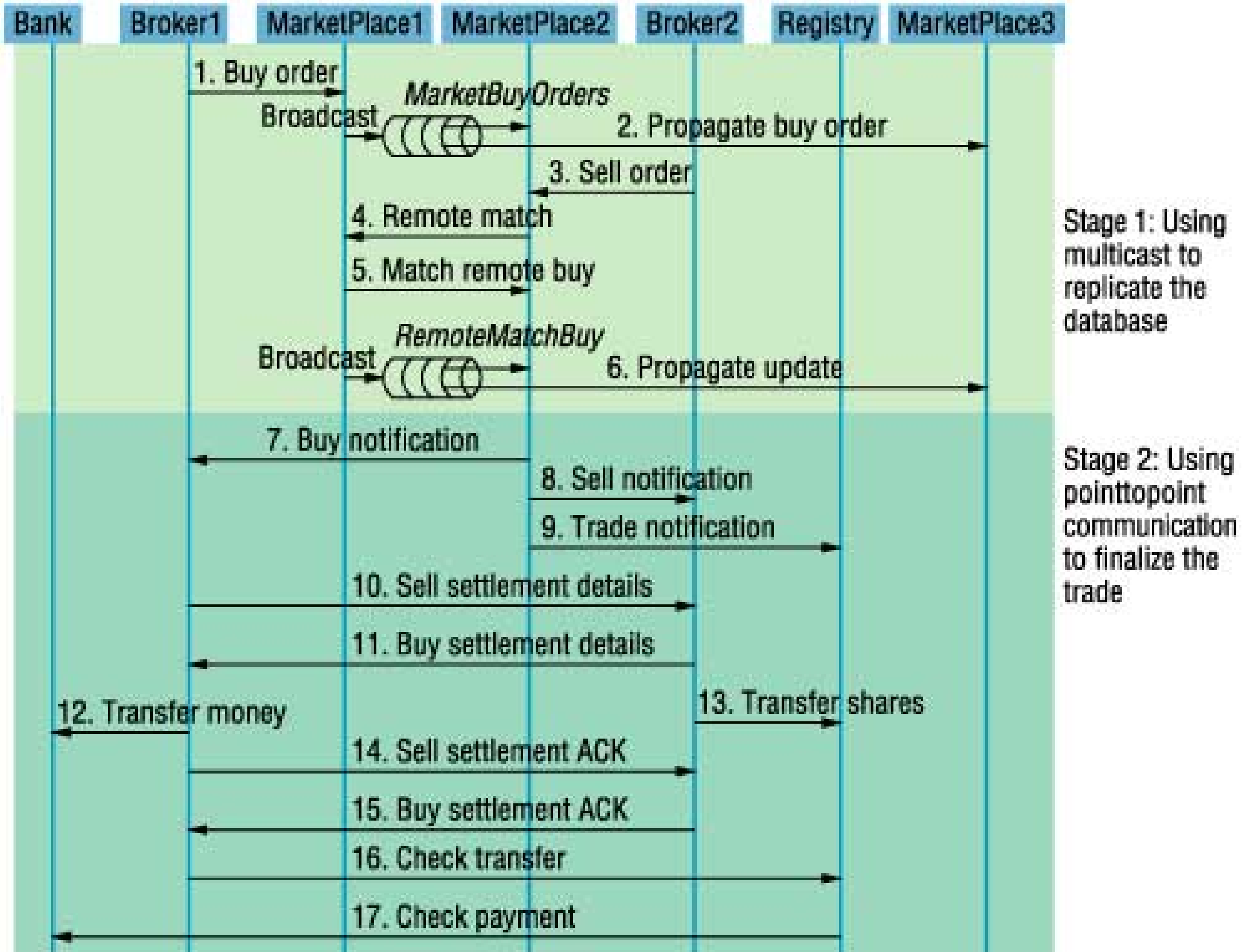

Figure 2. MarketSim basic business scenario.

We used asynchronous method calls to maximize the trading system's throughput. Message-passing orders aren't restricted, however, so we expect causal errors. MarketSim has an extensive method-call-logging system to trace messages between processes. If a causal error occurs, the log reflects it but the program doesn't stop running.

System architecture

MarketSim implements the entities shown in Figure 2 as independent

المنارة للاستشارات
www.manaraa.com

services on simple object access protocol (SOAP) servers. The entities communicate with each other by invoking SOAP method calls. Because of SOAP's distributed nature, a programmer can't observe any differences between calling a remote SOAP service and calling a local SOAP service. To achieve maximum throughput, we use concurrent multithreading to avoid waiting for SOAP calls to return. In the *point-to-point stage* (stage 2 of Figure 2), we use such asynchronous method calls extensively whenever the business logic allows.

We adopted the Java message service (JMS) mechanism to implement multicasting among marketplaces. Marketplaces use the MarketBuyOrder topic to broadcast buy orders, and the RemoteMatchBuy topic to broadcast order update information. Marketplaces both publish and subscribe to these two topics. When a new buy order arrives, the home market adds a record to its local database and sends the buy order message to MarketBuyOrder. Other marketplaces listen to the topic and add a record to their local databases when they receive a message. The system can't achieve atomic multicast because message delivery is only best effort. Similarly, marketplaces use RemoteMatchBuy to propagate matched orders and delete orders from local databases on receipt of a message.

The design has obvious flaws. Multithreaded concurrent calls will cause race conditions. Causal order violations will occur when race conditions conflict with business logic. Populating the database via JMS is obviously not secure, and messages could be dropped or received late. However, these potential disasters are exactly what we are interested in. Our goal is to observe and analyze how "wrong" our simulation can be.

Potential causal errors

Not all the messages in Figure 2 must be completed in the order shown. We identified causally ordered events by analyzing the

المنارة للاستشارات

www.manaraa.com

business logic. The program already naturally forces the orders of some events. For example, a `Sell Order` must be received before a `Remote Match` can occur, but the MarketPlace code enforces that. Because there are no restrictions on MarketSim's order, the order of some events is open. In particular, the following four causal orderings can cause a trade to fail:

A marketplace must receive a Propagate update before it can send Buy notification, Sell notification, or Trade notification messages.
A Buy notification must be completed before Buy settlement details starts;
An Sell notification must be completed before Sell settlement details starts.
A Trade notification must be received before a Transfer shares is received.

Vector time-stamp-logging system

Because MarketSim has few processes, classic Lamport vector time stamps do not add a lot of overhead to sent messages. MarketSim time stamps all method calls occurring between processes and all broadcast messages. MarketSim has no hidden channel. It generates a separate log file for each entity and a merged log file at each run. Analyzing the log file can give us an accurate picture of the system's global state at any give time. The global state reasoned from the log files matches the database state, further proving MarketSim's correctness.

Simulation

MarketSim is implemented as a simulation running on a single 1.8-GHz machine with 512 Mbytes of RAM. We used Tomcat Web Server 4.1.6, and JMS 1.3 running over SOAP server 2.3.1. For the first set of trials we ran the same setup 56 times. Each trial involved 20 clients, six brokers, three marketplaces, two banks, and two registries. To simulate real network conditions, we introduced a

المنارة للاستشارات

www.manaraa.com

random delay of 10 to 20 milliseconds for each message sent.

Discussion

Figure 3 is a graph of the failed trades as a percentage of successfully completed trades.

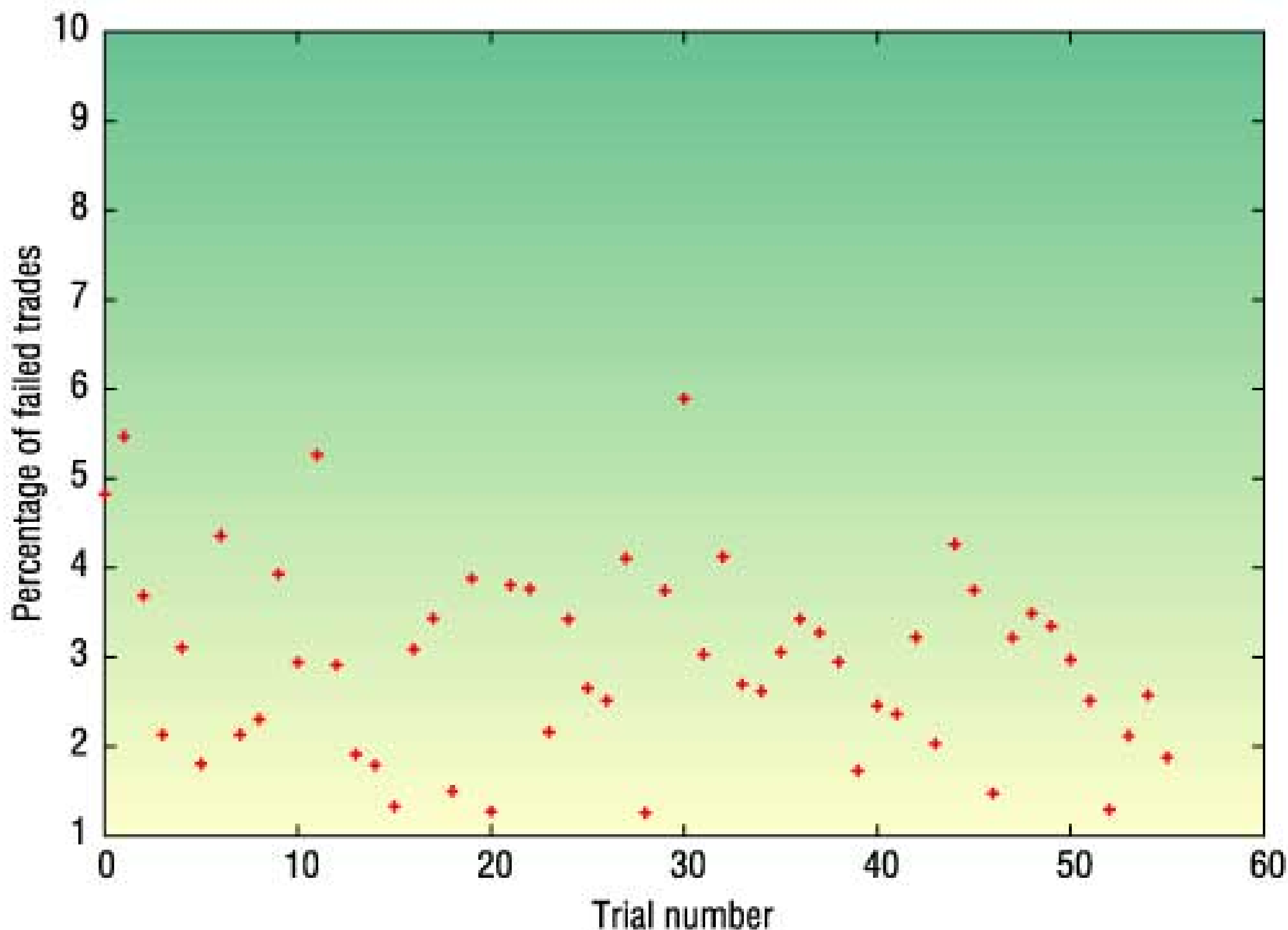

Figure 3. Percentage of failed trades per trial number.

As Figure 3 shows, there is some variation in the failure rate of trades in each trial. This is to be expected with random network delays. The variation, however, is not great and the percentage of failed trades averages to 2.97 percent over all runs with a standard deviation of 1.08 percent.

المنارة للاستشارات

www.manaraa.com

Is this a low number? The answer depends on the application. For some applications, any failure rate is unacceptable. But many applications, including MarketSim, can recover from a failed trade: if a trade fails to complete, it can be resubmitted.

Future work

The question now becomes one of tradeoffs. What is the overhead of introducing causal order into a system, particularly for the 97percent of successful transactions? We must contrast this with the cost of resubmitting the 3 percent of failed transactions. This is future work.

Also, we wish to perform more extended runs of the system, manipulating the delays in the simulation. We believe widely disparate network delays and processor speeds will lead to a much greater percentage of failed trades.

We would also like to extend MarketSim's functionality, making it a more realistic simulation, and introduce more complicated business logic. This will lead to greater understanding of the causal order errors that cause trades to fail and those that are irrelevant to the correct functioning of the system. We are also interested in extending this work to other business logic and collecting similar information.

Finally, after gathering this data, we expect to be able to define a classification system for possible ordering issues in distributed applications, with the goal of aiding the application developer in building distributed systems. Such a classification would show when the developer must be aware of causal ordering issues and when these issues can be ignored.

Acknowledgments

المنارة للاستشارات

www.manaraa.com

This work is based heavily on MarketSim, a stock market simulation written by Ryan Junee and Thomas Richards. Thanks also go to Alan Fekete, who generated many of these ideas.

References

1. D. Cheriton and D. Skeen, "Understanding the Limitations of Causally and Totally Ordered Communications," *Proc. 14th ACM Symp. Operating Systems Principles*, ACM Press, 1993, pp. 44-57.
2. J.H. Saltzer, D.P. Reed, and D.D. Clark, "End-to-End Arguments in System Design, *ACM Trans. Computer Systems*, vol. 2, no. 4, Nov. 1994, pp. 277-288.
3. K. Birman, "A Response to Cheriton and Skeen's Criticism of Causal and Totally Ordered Communication," *Operating Systems Rev.*, vol. 28, no. 1, 1994, pp. 11-21.
4. K. Birman and R. van Renesse, eds., *Reliable Distributed Computing with the Isis Toolkit*, IEEE CS Press, 1994.
5. Y. Amir, *Replication Using Group Communication over a Partitioned Network*, PhD thesis, Inst. of Computer Science, Hebrew Univ. of Jerusalem, 1995.
6. D. Dolev and D. Malki, "The Transis Approach to High Availability Cluster Communication,"*Comm. ACM*, vol. 39, no. 4, Apr. 1996, pp. 64-70.
7. D.L. Tennenhouse et al., "A Survey of Active Network Research," *IEEE Comm.*, vol. 35, no. 1, Jan. 1997, pp. 80-86.
8. M.J. Ranum, "A Network Firewall," *Proc. 1st World Conf. System Administration and Security*, 1992.
9. S. Maffeis, R. van Renesse, and K.P. Birman, "Horus: A Flexible Group Communication System, *Comm. ACM*, vol. 39, no. 4, Apr. 1996, pp. 76-83.
10. 10. L. Lamport, "Time, Clocks, and Ordering of Events in a Distributed System," *Comm. ACM*, vol. 21, no. 7, July 1978, pp. 558-565.

المنارة للاستشارات

www.manaraa.com

Nikki Lesley is a lecturer in the School of Information Technologies at the University of Sydney. Her research interests include theoretical properties of distributed systems, middleware technology, and pervasive computing. Contact her at nikki@cs.usyd.edu.au.

Holly Hou is an honors student in the School of Information Technologies at the University of Sydney. Her research interests include distributed systems and middleware technology. Contact her at Holly.Hou@honeywell.com.

Joydeep Mitra is an engineering student in the School of Electrical and Information Engineering at the University of Sydney. Contact him at jmit2685@mail.usyd.edu.au.

المنارة للاستشارات

www.manaraa.com