Real-Time Automatic Detection of Stuck Transactions

Diploma thesis subject of

Peter Hofer (0855349)

in cooperation with

Compuware Austria GmbH

September 2012

Abstract

The execution of business transactions in software typically involves multiple threads, processes and databases on different computing nodes, and communication between them. Problems in coordinating the use of shared resources or the exchange of data can cause these transactions to become stuck, causing user frustration and data loss, as well as failures and performance degradation in other transactions. The objective of this thesis is to identify common problem scenarios and to create a working prototype for detecting and diagnosing them to aid in preventing future occurrences.

Motivation

A *business transaction* in software is a sequence of operations performing a common task. In the simplest case, all operations of a transaction execute in a single thread on one computer. Usually however, multiple transactions run at the same time, and their execution involves several threads, processes and databases on different computing nodes and communication between them.

In such typical applications, access to common resources such as shared memory or database tables must often be restricted to one transaction at a time. When such synchronization is not in place, data can be lost or become corrupted. However, improper synchronization measures can lead to a transaction becoming *stuck,* preventing it from completing [1, 2, 3]:

Starvation occurs when a transaction is continuously denied access to a resource it needs to continue. For example, in an application that assigns resources solely based on the requesting transaction's priority, low-priority transactions will starve when higher-priority transactions continuously request needed resources.

Deadlocks happen to groups of transactions in which all transactions try to acquire resources that other transactions in the group hold. In the simplest case, one transaction holds resource r_1 and tries to acquire another resource r_2, but r_2 is held by another transaction which is trying to acquire r_1 at the same time.

Livelocks occur when transactions remain active (busy) without making progress. Sometimes this is the result of an attempt to resolve deadlocks: when the involved transactions fail to acquire a resource, they release their held resources, but then reacquire them in the same order, repeatedly ending up in the same situation.

Races are the result of missing synchronization when access to a resource should be coordinated or operations must be performed in a particular order. When a transaction misses an event from another transaction as result of a race, it can become stuck waiting for the event to happen.

Transactions can also become stuck when **waiting to read input data or write output data**. Common causes are congested networks, overloaded remote computing nodes, or defective hardware.

Stuck transactions can also simply be the result of **programming mistakes**. For example, unconsidered atypical input values or overflows can result in a transaction becoming stuck in an infinite loop.

A stuck transaction can have several negative consequences, such as:

Negative user experience when the stuck transaction leads to unresponsiveness in the user interface or user actions are not being carried out.

Data loss can ensue when a stuck transaction remains unnoticed.

Resources occupied by the stuck transaction, such as computing power, memory, threads and processes, files or network connections remain unavailable to other transactions. This can result in failures and performance degradation and also cause other transactions to become stuck.

Hence, a mechanism for detecting stuck transactions and diagnosing their causes constitutes a valuable tool for application developers and users.

Objective

The objective of this thesis is to create a working prototype implementation of real-time stuck transaction detection. This implementation will be able to detect different kinds of situations where a transaction has likely become stuck. When such a situation is detected, additional data is collected and analyzed to assess the circumstances. The user is notified and provided with an accurate analysis to help resolve the situation and prevent the problem in the future. The detection mechanism should be as unintrusive to the monitored application as possible and ideally require no prior configuration.

The prototype developed for this thesis will be able to detect stuck transactions in Java applications. It is expected that the general mechanics of the implementation also apply to other platforms such as Microsoft .NET and native applications.

The completed prototype will be integrated into *Compuware dynaTrace*[4], a performance analysis suite with the ability to monitor applications "end-to-end" across all its tiers.

Approach

Work on the thesis will be done in the following steps:

1. **Research** common reasons why transactions become stuck, how they behave in that situation, and how such a situation could be detected in (near) real-time. Also investigate and evaluate available software that has implemented this capability.
2. **Create sample code** modeling situations where transactions become stuck. The code could be introduced in a real-world application to get more realistic behavior. Observe the run-time properties the samples exhibit and determine which situations can realistically be detected automatically.
3. **Implement** a prototype which can automatically detect stuck transactions.
4. **Evaluate** the implemented solution and determine where detection works well, and where and how it could be improved. Measure the impact on observed applications.

Figure 1 shows the intended architecture of the prototype. An *agent* runs within the Java virtual machine of the application. It is responsible for monitoring the application for stuck transactions and forwarding relevant information to the server. The *server* collects and analyzes data from the agent and provides them to *clients*, which display them to users. For multi-tier applications, separate agents attach to each individual tier and transmit information to a single server.

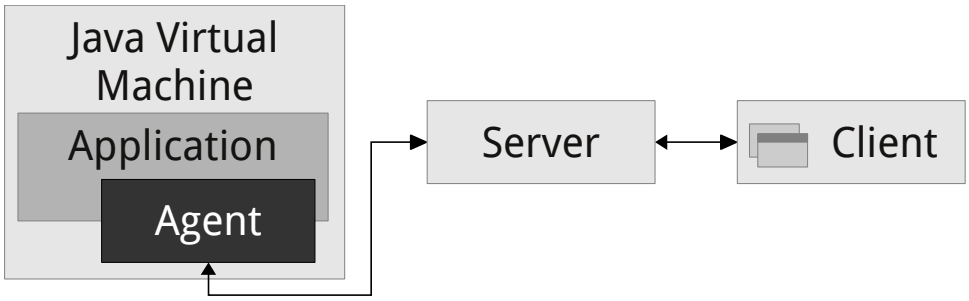

Figure 1: Intended Architecture

Some of the specific tasks the individual components will need to perform are:

The agent needs to detect when a transaction spends too much time waiting, but also when there is activity without progress. In order to diagnose locking problems, the agent needs to be aware of the locks held by individual threads. It should also be able to recognize and instrument locking code other than Java monitors, such as

ownable synchronizers[1] or the `wait()` and `notify()` methods of the `Object` class[2]. For detecting infinite loops, it could be necessary to instrument individual branch instructions.

The server, besides collecting and processing events, could detect stuck transactions or correlations between them on a high level using information from different agents.

The client should provide views and visualizations that specifically aid in comprehending stuck transactions.

References

[1] X. Song, H. Chen, and B. Zang, "Why software hangs and what can be done with it," in *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, pp. 311 –316, July 2010.

[2] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," *SIGARCH Comput. Archit. News*, vol. 36, pp. 329–339, Mar. 2008.

[3] D. Lea, *Concurrent Programming in Java: Design Principles and Patterns*. Reading, MA: Addison-Wesley, second ed., 2000.

[4] "Compuware dynatrace." `http://www.compuware.com/application-performance-management/dynatrace-enterprise.html`, Sept. 2012.

Related Work

[5] N. Nakka, G. Saggese, Z. Kalbarczyk, and R. Iyer, "An architectural framework for detecting process hangs/crashes," in *Dependable Computing - EDCC 5* (M. Dal Cin, M. Kaâniche, and A. Pataricza, eds.), vol. 3463 of *Lecture Notes in Computer Science*, pp. 103–121, Springer Berlin / Heidelberg, 2005.

[6] J. Burnim, N. Jalbert, C. Stergiou, and K. Sen, "Looper: Lightweight detection of infinite loops at runtime," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, (Washington, DC, USA), pp. 161–169, IEEE Computer Society, 2009.

[7] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: a dynamic data race detector for multithreaded programs," *ACM Trans. Comput. Syst.*, vol. 15, pp. 391–411, Nov. 1997.

[1] `http://docs.oracle.com/javase/7/docs/api/java/lang/management/LockInfo.html#OwnableSynchronizer`

[2] `http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html`

[8] K. Havelund, "Using runtime analysis to guide model checking of java programs," in *SPIN Model Checking and Software Verification* (K. Havelund, J. Penix, and W. Visser, eds.), vol. 1885 of *Lecture Notes in Computer Science*, pp. 245–264, Springer Berlin / Heidelberg, 2000.

[9] S. Bensalem and K. Havelund, "Dynamic deadlock analysis of multi-threaded programs," in *Hardware and Software, Verification and Testing* (S. Ur, E. Bin, and Y. Wolfsthal, eds.), vol. 3875 of *Lecture Notes in Computer Science*, pp. 208–223, Springer Berlin / Heidelberg, 2006.

[10] M. Carbin, S. Misailovic, M. Kling, and M. Rinard, "Detecting and escaping infinite loops with jolt," in *ECOOP 2011 – Object-Oriented Programming* (M. Mezini, ed.), vol. 6813 of *Lecture Notes in Computer Science*, pp. 609–633, Springer Berlin / Heidelberg, 2011.

[11] D. Marino, M. Musuvathi, and S. Narayanasamy, "Literace: effective sampling for lightweight data-race detection," in *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, (New York, NY, USA), pp. 134–143, ACM, 2009.