Technical Report

UCAM-CL-TR-633
ISSN 1476-2986

Number 633

UNIVERSITY OF
CAMBRIDGE

Computer Laboratory

On deadlock, livelock, and forward progress

Alex Ho, Steven Smith, Steven Hand

May 2005

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

ISSN 1476-2986

On Deadlock, Livelock, and Forward Progress

Alex Ho, Steven Smith, and Steven Hand
Computer Laboratory, University of Cambridge

1 Introduction

Many algorithms have been developed for detecting *deadlock* in distributed systems. Most assume the ability to identify *a priori* each lock use, either automatically or by requiring manual annotation. On the other hand, relatively little research has gone into *livelock* detection, possibly due to confusion about the term itself—in the past, the computing literature has used the term livelock inconsistently to mean starvation, infinite execution, or simply the failure to maintain liveness.

In this paper we propose a general dynamic framework for detecting deadlock and livelock in distributed systems. We unify both of these undesirable occurances under the general term *standstill*: a system is at a standstill if no forward progress is being made. This is an inherently domain-specific concept that ranges from simple circular synchronization problems (traditional deadlock) to application-level and even user-level notions of progress.

For example, consider the following 'reliable' data transfer protocol. The message is broken down into individual packets, which are sent in groups via UDP to the destination. As each group is received, an acknowledgement packet is returned to the sender indicating which packets were not received and hence need to be resent. The absence of a timeout is a bug. In the pathological case where all packets are lost, the receiver is stuck waiting for a packet and the sender is stuck waiting for an acknowledgement. The system is at a standstill.

We have developed a four stage framework to detect these—and more complex—situations. First, we obtain or derive a set of predicates on the distributed systems' execution state. Each atom of a predicate refers to a particular process and will typically concern the location of the program counter (e.g., "return from `service_request()`"), the value of a variable (e.g., "`x > 25`"), or the triggering of an I/O event (e.g., "received network packet containing `0xEABD`"). Building on these basic elements, we can construct predicates that define one particular state of the distributed system.

Second, we dynamically monitor the system to determine when states are entered. There is a trade-off between precise but potentially intrusive monitoring and low overhead but less accurate schemes. Third, we observe state transitions and detect lack of progress within a fixed time: in essence we build a 'watchdog timer' to recognize application-level standstill. Finally, if standstill is detected, we attempt to repair the situation and, if required, notify the user.

In the remainder of this short paper we first briefly survey related work, elaborate on our notions of *forward progress* and *standstill*, describe our algorithm in more detail, present an initial prototype, and discuss future research challenges.

2 Related Work

Previous solutions for deadlock and livelock detection can be divided into several broad categories. Distributed deadlock detection algorithms require knowledge of each lock acquisition and release and do not handle any form of livelock [7, 14].

Dynamic analysis tools monitor a running program. Most search for a loop in a graph of allocated resources and pending requests [10, 13] but do not generally extend to distributed systems.

Model checking tools, such as Spin [6] and TLA [8], utilize a model of the distributed system. Using formal methods, the state space is verified exhaustively. One drawback to model checking techniques is the abstraction imposed by the model; differences between the model and program implementation may result in undetected flaws.

Static analysis tools analyze the application source code. For example, RacerX [4] uses source code annotations of lock acquisition and release operations to detect potential deadlocks in multi-threaded systems. It is unclear how effective static analysis could be for highly nondeterministic distributed systems.

3 Forward progress... or lack thereof

We state informally that an application makes *forward progress* when it performs useful computation towards termination[1]. As long as the program is executing as the programmer intended, it is making forward progress. This clearly requires detailed knowledge of the application's intended behavior.

Conversely, *standstill* represents the lack of forward progress. Both deadlocked and livelocked systems are at a standstill.

3.1 Livelock

Although numerous inconsistent definitions of livelock have been used in the literature, the term usually connotes one of the following:

Starvation: Systems with a non-zero service cost and unbounded input rate may experience starvation. For example, if an operating system kernel spends all of its time servicing interrupts then user processes will starve [11].

Infinite Execution: The individual processes of an application may run successfully, but the application as a whole may be stuck in a loop [15]. For example, a naïve browser loads web page A that redirects to page B that erroneously redirects back to page A. Another example is a process stuck traversing a loop in a corrupted linked list.

[1]Or in the case of a server or daemon, computation towards the completion of a service request.

Breach of Safety Properties: The safety property of distributed systems states that the application will not perform an incorrect action or enter an undesirable state[2] [12]. By adding a temporal attribute to the application state, we say that if the program does not make forward progress within some time bound it is livelocked. For example, if the temporal rule "a response is sent for every request within 10 seconds" fails then the server is deemed to be at a standstill.

Creating the appropriate liveness specifications for a given application requires detailed domain knowledge about the program's intended behavior and internals of its implementation.

3.2 Deadlock

A set of threads is deadlocked if each thread is waiting for an event that can only be generated by another thread in the set [16]. The event is usually the release of a resource protected by a lock, and therefore deadlock reduces to the inability to acquire a lock.

The choice of lock implementation depends on the application's resource usage patterns. Operating systems use fine-grained spinlocks to protect data held for short periods of time. A database may delegate lock management to an external distributed lock manager provided by the operating system [3].

Although the internals of each lock implementation differ, on close inspection they all behave in a similar fashion when a deadlock occurs. The application's individual processes are still running, but execution is confined to the lock implementation code. In this case, forward progress is the ability to acquire the lock, and the lack of forward progress is deadlock.

[2]The associated liveness property states that the program eventually performs a correct action. This only specifies that the action will complete after an infinite sequence of steps and does not provide a fixed time bound; it is not desirable to define a livelocked system using the liveness property.

```
    void transmit (msg)
    {
      for (idx = 0; idx < msg.count; ) {
        for (loop = 0; loop < batch_size &&
             idx < msg.count; loop++, idx++) {
          write (socket, msg[idx]);
        }
(1)
        read (socket, ack);
        /* resend missing pieces */
      }
    }
```

Figure 1: Sample client code to transmit a message that has been subdivided into `msg.count` pieces.

4 Standstill Detection

As described previously, our general framework for detecting distributed systems at a standstill involves predicates, dynamic state detection, and state transitions. First, predicates are generated for the application, with each predicate characterizing a state of the computation. In addition, a set of temporal rules indicating valid compositions of the states is created. The state predicates and temporal rules are collectively called a *liveness signature.*

Next comes dynamic *detection* of individual states and the *composition* of these using the liveness signatures to determine if the system is at a standstill. Finally, a mechanism is provided to automatically *repair and recover* the distributed system.

4.1 Liveness Signatures

A liveness signature is a partial specification of application states that are significant in determining whether a program is making forward progress, and the valid transitions between those states. For example, consider the code shown in Figure 1. The trivial state predicate `pc = (1)` defines the state *batch sent* where the client has successfully transmitted a batch of packets and is awaiting an acknowledgement.

Liveness signatures are not limited to synchronization operations; they are indicative of desired application behavior and will often specify a higher-level liveness metric such as "the packet has been acknowledged" or "the web page has been rendered".

A state predicate may encompass any aspect of the environment in which processes execute: values in the processes' memory or registers (including program counter), the state of the underlying OS, the contents of disks and other peripherals, and packets within any intermediary network.

4.2 Detection

The first task of standstill detection is to recognize fulfilled state predicates. As the application executes, the system must recognize when the program enters a state described by an attached predicate. There are different ways to do this depending on the placement of the signatures.

If the signatures appear as code annotations then an additional co-routine and runtime library can monitor the program and signal whenever a liveness state has been reached. Alternatively, an external agent can watch the target application and trigger a signal when a state is entered. Hybrid solutions involving dynamic code modifications are also possible.

State predicates are built from two types of atomic triggers. Value conditions depend on the contents of an object, such as a memory location, disk block, or network packet. Temporal conditions indicate events such as a certain line of code being executed or a network packet being received. By analogy with debugging, these are named respectively *watchpoints* and *breakpoints.*

4.3 Composition

Each state predicate only indicates that the program is making forward progress at an instant in time. The developer must specify a set of rules which specify the valid transitions between the liveness states and, optionally, time bounds on those transitions. For example, Figure 2 illustrates that the state *ack received* must

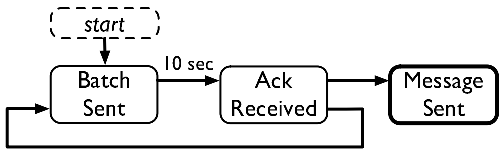

Figure 2: State transition map

be visited between *batch sent* and *message sent* and that this must occur within 10 seconds of the previous visit to *batch sent*.

The same diagnostic mechanism used to detect the presence of liveness states is used to verify the transitions between the states. Either a co-routine or an external agent can use a simple watchdog timer. When a liveness state is reached the algorithm checks that the liveness signature is valid given the history of previously seen states and, if so, resets the timer. If the timer fires before another valid state is reached, the system is deemed at a standstill.

4.4 Recovery & Repair

If the monitor detects a system at a standstill, it can continue to trace the target processes and display diagnostic data to the developer. By single stepping the program, it is possible to determine precisely which lines of code are executing, inspect the call graphs for each process, list which variables are being accessed and modified, and investigate network and disk I/O operations. In combination this data can aid a developer in uncovering the cause of the problem.

In a production system, the monitor can collect the above debug information for later analysis and can also attempt to automatically "repair" the application. For example, by periodically checkpointing the distributed application, we can subsequently time-warp a system at a standstill to a previous state and allow it to run anew. The ensuing run may complete successfully if the original error was the result of non-deterministic execution.

5 Prototype

Our prototype is based on the PDB debugger [5], which has been implemented within the Xen virtual machine monitor [1], to check liveness signatures in distributed applications. PDB's unique placement within a virtual machine monitor enables it to synchronously control all of the processes of a distributed application and access the entire execution environment including operating systems, system libraries, and virtual devices.

5.1 Liveness Signatures

Our prototype uses a Python client to issue commands over TCP to a daemon process running in a Xen control VM. The daemon is responsible for interacting with the target processes, and the client is responsible for detecting the liveness signatures.

The client allows a developer to implement liveness signatures in Python. Considering the dining philosophers example shown in Figure 3, the following code detects whether or not the system is at a standstill:

```
p1 = context vmid1 pid1
p2 = context vmid2 pid2
p3 = context vmid3 pid3

def process_is_blocked(p):
  return p.blocked()
    and (p.at_pc((2)) or p.at_pc((3)))

def standstill():
  return process_is_blocked(p1)
    and process_is_blocked(p2)
    and process_is_blocked(p3)

Watcher.subscribe(standstill, event_consumer)
```

`p.blocked` indicates whether the OS thinks that the process is blocked, and the final line notifies the user when the standstill condition becomes true.

5.2 Overhead

We have performed an initial evaluation of our prototype, both to measure the worst-case

```
      while (true)
      {
        ponder()
(2)     get left fork()
(3)     get right fork()
(4)     eat()
        release left fork()
        release right fork()
      }
```

Figure 3: Dining Philosophers Pseudo-Code

overhead and to explore its use in a more realistic setting. All tests were performed with Xen 1.3 on a 2.4 GHz Intel Xeon, with 512MB allocated to the Xen control VM and 32MB to each user VM where the tests ran under Linux 2.4.27. For each test, the average and standard deviation of multiple runs is provided.

To test worst case absolute overhead, we wrote a small microbenchmark that simply invokes the signature detection engine 10,000 times with either a watchpoint or a breakpoint. The aggregate cost was 118.5 ± 0.5 sec in the former case and 82.8 ± 0.4 sec in the latter, a fairly substantial overhead.

This is mainly due to trapping into the Xen VMM, context switching into the administrative VM, and forwarding the events across the network to the remote Python client. We took this approach in our prototype since it allows for considerable development flexibility. We anticipate a significant performance improvement by pushing more of the detection logic closer to the processes being monitored.

As a more realistic measurement of overhead for the entire standstill detection framework, we ran the simple dining philosophers example with 3 philosophers, each in its own VM. The processes synchronized by sending messages across TCP sockets. We varied the amount of time spent eating on line **(4)** and measured the number of philosophers that got to eat each second.

eat time (msec, uniform distribution)	Xen	Xen + PDB
	(number of meals per sec)	
$0 < t \leq 125$	8.55 ± 0.88	8.97 ± 0.69
$0 < t \leq 250$	6.13 ± 0.09	5.47 ± 0.41
$0 < t \leq 500$	3.10 ± 0.05	3.06 ± 0.05
$0 < t \leq 1,000$	1.67 ± 0.09	1.50 ± 0.07

Liveness signature detection imposes at most a 10 to 15% overhead. The higher rate at 125 msec are most likely due to scheduling artifacts.

6 Ongoing Research Challenges

There are numerous refinements that can be made to each of the four phases of the algorithm. Here we briefly discuss some challenges we have identified.

6.1 Signatures

Signatures could be extended to include application-specific data such as performance counters [2] and other event sources. Furthermore we are interested in determining how much benefit might be gained from additional historical information such as a program counter trace [9].

In addition, our present implementation requires the developer to create liveness signatures manually based on debugger primitives relating to program state. The signatures should be incorporated into a high level language to facilitate their use. We are also interested in investigating the automatic generation of signatures via supervised learning.

6.2 Composition

The present architecture currently checks for forward progress every time the application triggers a liveness signature. While this is useful when analyzing potentially buggy software, a significant performance slowdown is incurred. Instead, the agent can periodically check to see if the target is at a standstill. This results in a trade-off between detection precision and overall performance.

6.3 Recovery & Repair

We are actively working on building support for "time warp" repair—returning to a safe checkpointed state—in the case of standstill detection. It may also be possible to repair by directly changing the system state, such as triggering retransmission of a packet that can enable forward progress.

7 Conclusion

Deadlock and livelock can happen at many different levels in a distributed system. We have proposed unifying both concepts and building a framework capable of detecting the lack of forward progress. Our initial prototype, although simple, can easily solve traditional deadlock problems even where synchronization is via a custom network protocol; however, many interesting research challenges remain.

References

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pp 164–177, Oct. 2003.

[2] B. Cantrill, M. Shapiro, and A. Leventhal. Dynamic Instrumentation of Production Systems. In *Proceedings of the USENIX 2004 Annual Technical Conference*, pp 15–28, Jun. 2004.

[3] Compaq Computer Corporation. *OpenVMS Programming Concepts Manual*, 7.3 ed., Jun. 2002.

[4] D. Engler and K. Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pp 237–252, Oct. 2003.

[5] A. Ho, S. Hand, and T. Harris. PDB: Pervasive Debugging with Xen. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, Nov. 2004.

[6] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[7] E. Knapp. Deadlock Detection in Distributed Databases. *ACM Computing Surveys*, 19(4):303–328, Dec. 1987.

[8] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[9] J. Larus. Whole Program Paths. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pp 259–269, 1999.

[10] T. Li, C. Ellis, A. Lebeck, and D. Sorin. Pulse: A Dynamic Deadlock Detection Mechanism Using Speculative Execution. In *Proceedings of the USENIX 2005 Annual Technical Conference*, Apr. 2005.

[11] J. Mogul and K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-Driven Kernel. *ACM TOCS*, 15(3):217–252, Aug. 1997.

[12] S. Owicki and L. Lamport. Proving Liveness Properties of Concurrent Programs. *ACM TOPLAS*, 4(3):455–495, Jul. 1982.

[13] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multi-threaded Programs. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pp 27–37, Oct. 1997.

[14] M. Singhal. Deadlock Detection in Distributed Systems. *IEEE Computer*, 22(11):37–48, Nov. 1989.

[15] K. Tai. Definitions and Detection of Deadlock, Livelock, and Starvation in Concurrent Programs. In *Proceedings of the 1994 International Conference on Parallel Processing*, vol. 2, pp II:69–II:72, Aug. 1994.

[16] A Tanenbaum. *Modern Operating Systems*. Prentice-Hall, 2001.