Protecting Applications Against Heisenbugs

Chris Hobbs, Kernel Developer
QNX Software Systems
chobbs@qnx.com

Abstract

Virtually-synchronous replication provides a mechanism that allows developers of mission- and safety-critical applications to reduce the impact of elusive and non-reproducible bugs—commonly known as Heisenbugs—in their applications. This approach requires little or no modification to existing application code, and significantly improves the software availability.

Introduction

All developers of sophisticated applications are familiar with non-reproducible bugs: bugs that occasionally cause crashes during testing or—worse—in the field. Repeating the test that failed does not reproduce the bug, because it was caused by unusual transient conditions that cannot be artificially recreated. Even if the crash is “reasonably” reproducible, adding debug code often changes its nature, or causes it to disappear.

The term Heisenbug was coined by Jim Gray or Bruce Lindsey (the origin is a little obscure) to refer to these elusive, non-reproducible bugs\(^1\). These bugs can be contrasted with Bohrbugs, the hard bugs that manifest themselves every time a specific section of code is executed with particular input. The references are, of course, to Niels Bohr’s billiard-ball model of the atom and Werner Heisenberg’s model where the act of observing alters the subject of observation.


Figure 1: A computer program

In general, Bohrbugs are amenable to conventional debugging techniques, and are relatively easy to track down and fix. Heisenbugs, in contrast, are typically caused by subtle timing interactions and disappear when the code is run in debug mode. In many cases, Heisenbugs may be seen once during system testing but may never again be reproduced.

Some early research by Jim Gray indicates that Bohrbugs tend to diminish over time as the software is used. Anita
Borr found, however, that they tend to be replenished when a new release or an upgrade is issued. Heisenbugs, on the other hand, tend not to diminish with time, and so eventually become the major cause of failure in mature software.

Heisenbugs are a particular threat to mission- and safety-critical systems; they are “time-bombs” known to be in the application but impossible to track down or remove.

This whitepaper proposes a mechanism which an application can use to reduce or even eliminate the impact of latent Heisenbugs. It is structured as follows:

- “The Increasing Impact of Heisenbugs” briefly explains why Heisenbugs are increasingly important in embedded software applications.
- “Virtually-Synchronous Replication” describes the underlying technology on which the proposed solution to message-timing Heisenbugs is based.

The Increasing Impact of Heisenbugs

In the last decade, software has become an increasingly central element in mission- and safety-critical embedded systems. These systems need to have sufficient availability (to respond when invoked) and reliability (to respond correctly when invoked) to maintain the required level of integrity of the system.

Over the same period, the software in many of these applications has moved from singled-threaded, run-to-completion code on an “executive” program running on a simple micro-controller to priority-scheduled, multi-threaded code running on synchronous multi-processing (SMP) architectures. Multi-threaded, SMP environments bring significant performance advantages to many applications; but they also bring subtle—and not insignificant—threats.

Figure 1 above illustrates the difference between a Heisenbug and a Bohrbug. A computer program is fundamentally a mechanism for reacting to input conditions (demands made upon it) and performing some actions. The input conditions are normally extremely complex, but for any program it is theoretically (although not practically) possible to define a multi-dimensional input space with one dimension per input variable. A simple, single-threaded program maps points into actions in this input space. Some of these actions are correct (the program worked) and some are not correct (the program has a bug). Fixing the bug consists of reconstructing the point in the input space that caused the incorrect behavior, then correcting the program’s response to this point.

In practice, this model is too simplistic. As Figure 1 illustrates, the same point in the input space may be reached on different “trajectories” (paths in time). The program may react differently, not only depending on the point in the multi-dimensional space, but also on the trajectory that brought it there. Multi-threading and SMP both provide significantly more degrees of this type of temporal flexibility.

Some trajectories passing through a particular point cause a Heisenbug while others are harmless. It is this observation that both explains the problem that needs to be resolved and justifies the technique for solving the problem.

---

2 There is a further category of bug, the SchrödInbug, named after Erwin Schrödinger, which refers to a bug that manifests itself when someone reading the source code notices that the program never should have worked in the first place. The program then stops working until fixed!

3 It would, of course, be possible to extend the dimensionality of the input space to include time, but this becomes even more difficult to imagine!
System designers and programmers moving out of the run-to-completion world to write their first multi-threaded application often find themselves victims of multiple trajectories passing through the same point in the input space. They also too often find that they have not fully covered all temporal possibilities: “what happens if thread A is pre-empted while executing B, and thread Z happens to run before A is resumed and makes use of the Y data structure that B was modifying …?”

As a programmer becomes more experienced these simple race conditions become less common, but the combinatorial complexity of most systems means that they are rarely eliminated. Moving to an SMP environment, where threads can be running simultaneously, introduces further temporal subtlety: the programmer can no longer assume that because thread A is running thread B cannot also be running.

Thus, SMP systems are environments ideally suited for the appearance—and disappearance—of Heisenbugs. Unfortunately, conventional analysis of system availability and reliability based on techniques used in the analysis of hardware failures (Markovian modeling, M:N sparing, etc.) is not appropriate for these systems where software is the dominant source of failure. This fact has been noted in numerous articles, but warnings have not always been heeded\(^4\).

**Virtually-Synchronous Replication**

In the early 1990s, Ken Birman, Robbert van Renesse and others developed techniques and protocols for maintaining membership lists of groups of processes, and for distributing messages in defined order to members of a group. The purpose of this work was to provide high availability for a system in the face of the primary forms of failure in those days—network and hardware failures. A consequence of this form of operation was protection against Heisenbugs. In 1996, when Birman’s *Building Secure and Reliable Network Applications*\(^5\) was published, solving Heisenbugs was seen as a secondary consequence of their work; Birman only mentions it in passing in chapter 15.

In a multi-threaded, SMP-based system, the possibility of hardware failure must of course be considered in the system design. However, because in these systems software error is the most common source of failure, designs that are resilient to Heisenbugs have taken on a new importance. One solution is virtually-synchronous replication.

---

\(^4\) See, for example, Borr, Andrea and Carol Wilhelmy. (1994) “Highly Available Data Services for UNIX Client/Server Networks: Why Fault-Tolerant Hardware Isn’t the Answer” in *Lecture Notes in Computer Science*. New York: Springer.

general, they arise from the hardware-centric thinking mentioned earlier.

The basic problem is illustrated in Figure 2a. Here consumers (sometimes called “clients”) require some service from a provider (sometimes called a "server"). These consumers construct and send a request message to the provider, which should respond appropriately. The obvious problem with this architecture is that, should the provider fail because of a Heisenbug (or even a Bohrbug), the clients will stop receiving service.

Architecture 2 (Figure 2b) is rarely seen today, but it was at one time very popular in telecommunications applications; it has multiple copies of the provider, each copy running on separate hardware in synchronized (locked) step. While this architecture offers some protection from hardware failure, it does not protect against any form of software failure because if a software bug is encountered all replicas may fail simultaneously.

Architecture 3 (Figure 2c) is more common: the provider is replicated and, at any time, one replica is active (responding to consumer requests) while the other is passive. If the active provider fails—due to a Heisenbug, for instance—then the (logical) switch is thrown and the passive provider becomes active. This mechanism is a complex one, however, and, unfortunately, it rarely works correctly. The problems it must solve include ensuring that:

• the passive provider is functioning and is able to accept the load should the active provider fail; silent failure of the passive provider is the main cause of outage in many systems.

• the passive provider is kept up-to-date—this necessitates an interface from the active to the passive provider

Since the passive provider must be up-to-date at whatever time the switch is thrown, this interface is complex and requires significant extra code. It therefore provides a hiding place for more bugs.

• the switch is thrown when needed and not thrown prematurely

This requirement depends on a clearly-defined mechanism for determining that the active provider has indeed failed. This mechanism requires additional software (the cause of more bugs!) to make the decision and throw the switch.

A Critical Observation
Jim Gray observed in the late 1980s that none of this complexity is required to allow a system to recover from a Heisenbug: simply repeating the same request will almost certainly work!

The solution to the Heisenbug problem lies in the nature of Heisenbugs themselves. Because a Heisenbug only manifests itself under very specific and elusive conditions (which cannot be replicated) the next time the same point is met in the system's demand space, the trajectory will be different, and the bug will probably not manifest itself.

Temporally Serial and Parallel Replication
The proposal in this paper is based on Gray's observation: running the same request again on the same
code will almost certainly work. This insight provides two mechanisms to protect a provider against a Heisenbug:

- **Temporally-serial** — capture and store the consumer’s request. If the provider fails, then restart it and re-apply the request. If the problem was a Heisenbug, then it will almost certainly succeed on the second invocation.

- **Temporally-parallel** — present the consumer’s request to two or more replicas of the provider virtually synchronously (the precise meaning of this term is given below). If a Heisenbug hits one replica, it is unlikely to hit them all.

The first of these mechanisms can be supported by a high-availability manager (see box on this page). The second mechanism is described in more detail below.

The QNX Neutrino High-Availability Manager monitors processes and restarts them if they have failed. Assuming that a Heisenbug will not occur twice in a row, this protocol provides an effective recovery from many failures caused by these bugs. The QNX Neutrino RTOS microkernel architecture ensures that such failures have minimal impact on the overall system.

The replication protocol then operates as follows (see Figure 3):

1. Consumers send requests to what they believe to be the unique provider. They are unaware that the provider is replicated, and so require no modification to operate in this environment.

2. Middleware distributed with the provider determines a delivery order for the requests appropriate to the application (see “Ordering” below).

3. The requests are replicated by the middleware and delivered to all replicas of the provider.

4. Each provider is unaware of other replicas (and so, in the simplest case, requires no modification to operate in this environment) and simply carries out its operation—calculating a result, updating a database, and so on.

5. Each provider responds as it normally would to the request. These responses are caught by the middleware and all but one (normally the first) are discarded. The one response is returned to the consumer.

Many variations to this basic theme can be introduced for performance or security reasons. In this simplest form, no changes are needed to consumers or providers (except a synchronizing interface if the provider state needs to be initialized). Since the same requests are delivered to all providers in an appropriate order, the providers always pass through the same sequence of states, although at different times.

The various replicas are not running synchronously (only “virtually synchronously”), so each will be running in its own timing and threading environment. This means that, although some replicas may fail because of a Heisenbug, it is unlikely that they will all fail; or if they do all fail, the cause is a Bohrbug. When a provider does fail, then it is restarted and resynchronized, and it becomes a member of the group again, all without the consumers being aware of any problems.

**General Design**

The basic principle of virtually-synchronous replication is illustrated in simplified form in Figure 3. For clarity, consumers are shown as distinct from providers; although in practice there is generally no distinction because in most systems a process is rarely purely a consumer or purely a provider.

With virtually-synchronous replications, a provider is replicated \( N \) times and the replicas are run on various processors. It is advantageous that more than one replica runs on one processor, and that replicas run on more than one processor. These processors might be cores within an SMP processor or processors connected via a network.
Of course, the magic is in the middleware, which is preferably written purely in application space, either as a separate server or as code integrated with each replica of the provider. The purpose of this middleware is threefold:

- to maintain the membership list of the groups to which the various providers belong

As Tushar Deepak Chandrat et al. demonstrated in their famous paper “On the Impossibility of Group Membership” published by in 1996, this is logically impossible, but algorithms exist for it to be achieved sufficiently well in the real world, particularly in embedded environments.

- to maintain the ordering of the incoming requests to the various replicas of the providers

Various algorithms exist for the interactions required to achieve this, some relying on a “master” and a process for handling the failure of the master, others relying on a totally-distributed algorithm. Different applications place different requirements on the strictness of the ordering.

- to (re)synchronize providers when they join a group by extracting the current state from an existing group member and providing it to the newcomer.

**Ordering**

The integrity of an application relies on the ordering of the requests passed to the providers. The types of ordering (in increasing strength) that are normally used are as follows:

**No Order** — Messages may be delivered to providers in any order.

This ordering would be adequate for consumers making read-only enquiries on an unchanging database: the answer to a query does not depend on the ordering of the query relative to other queries, and there is no state change in the provider.

FIFO Order — If a consumer sends message A before message B, then any provider that receives both A and B will receive A before B.

This ordering would be suitable for a consumer updating a private database (a database not accessible to other consumers). Clearly, if an update is sent before a query, then it is essential that all database providers receive the update before the query.

Causal Delivery Order — If a provider (now acting as a consumer) sends message B after receiving message A, then message A will be delivered before message B to all providers that receive them both.

This ordering would be needed if an update to a database causes another message to be triggered. It might be inappropriate for any provider to receive the consequential message before updating its copy of the database.

Totally Ordered Delivery — If one provider receives message A before message B, then all providers that receive these two messages will receive A before B.

This ordering would be needed when multiple consumers are operating in parallel on databases, carrying out both reads and writes. If one database provider receives update A before update B (coming from different consumers), then it is essential for consistency that all database providers receive the updates in that order.

In any application, the system designer would select the lowest level of suitable message ordering, since the increasingly sophisticated orderings bring performance degradation with them.

Tradeoffs

A performance penalty has to be paid when using virtually-synchronous replication: the tradeoff parameters are actually threefold, as shown in Figure 4. Generally system performance, data consistency and system availability are in conflict: for example, it is easy to design an application with very short response times if the system can tolerate data inconsistency and high levels of unavailability. Similarly, it is easy to create an application with extremely high availability if data consistency and performance may be sacrificed.

Traditionally, the form of replication suggested in this paper has provided high availability and reasonable data consistency at the expense of performance. The actual impact on performance depends largely on the number of provider replicas that are required to meet the availability goals of the system, and on the ordering guarantees required by the application.

Conclusion

When designing Heisenbug-resilient applications implementing virtually-synchronous replication, developers should consider the following three requirements:
• A synchronous, message-passing form of inter-process communication. An OS that uses this form of inter-process communication for intra- and inter-process messaging can provide a natural and consistent interception point for all messages, enabling replication to be carried out cleanly, as well as allowing existing applications to be ported to the virtually-synchronous design with minimal effort.

• A local- (rather than wide-) area network of processors with a low transmission failure rate. This requirement is met particularly well by different cores within the same SMP architecture, and can be met reasonably well by a system operating across a LAN.

• A means of restarting failed processes. The OS should support a reliable high availability manager, preferably in an architecture that minimizes the impact on the system of the failure of any component.

• An OS that allows the implementation of the necessary middleware in the application space rather than in the kernel space for greater system resilience.

 Acting on Gray’s observation made over two decades ago that the solution to Heisenbugs is simply to repeat the request that failed, developers can use virtually-synchronous replication on an appropriate OS to design applications that are resilient to these bugs—bugs that by their nature are so difficult, if not impossible, to identify and remove before product shipment.

About QNX Software Systems
QNX Software Systems is the leading global provider of innovative embedded technologies, including middleware, development tools, and operating systems. The component-based architectures of the QNX® Neutrino® RTOS, QNX Momentics® Tool Suite, and QNX Aviage® middleware family together provide the industry’s most reliable and scalable framework for building high-performance embedded systems. Global leaders such as Cisco, Daimler, General Electric, Lockheed Martin, and Siemens depend on QNX technology for vehicle telematics and infotainment systems, industrial robotics, network routers, medical instruments, security and defense systems, and other mission- or life-critical applications. The company is headquartered in Ottawa, Canada, and distributes products in over 100 countries worldwide.

www.qnx.com

© 2010 QNX Software Systems GmbH & Co. KG, a subsidiary of Research In Motion Limited. All rights reserved. QNX, Momentics, Neutrino, Aviage, Photon and Photon microGUI are trademarks of QNX Software Systems GmbH & Co. KG, which are registered trademarks and/or used in certain jurisdictions, and are used under license by QNX Software Systems Co. All other trademarks belong to their respective owners. 302152 MC411.80