Fast Failure Detection
in Distributed Software Radio Applications

by

Stanley Hu

B.S., Massachusetts Institute of Technology (2000)

Submitted to the Department of
Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2001

© Stanley Hu, MMI. All rights reserved.

The author hereby grants to MIT permission to reproduce and
distribute publicly paper and electronic copies of this thesis
documents in whole or in part.

Author

Department of Electrical Engineering and Computer Science

May 23, 2001

Certified by

Stephen J. Garland
Principal Research Associate
Supervisor

Accepted by

Arthur C. Smith
Chairman, Department Committee on Graduate Students
Fast Failure Detection
in Distributed Software Radio Applications

by

Stanley Hu

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2001, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis describes and evaluates a mechanism for quickly detecting failures in
distributed software radio applications that run on general-purpose processors and
operating systems. Machines monitor their connected machines with rapid keepalives
and multicast a failure report if some number of keepalives have not arrived. Using
standard Linux system calls, this system can accurately detect a failure in 50 ms.
When the system is stressed with competing processes, failure detection times rise to
approximately 300 ms. Achieving faster detection is currently limited by two factors:
the 10 ms interval with which Linux can send and receive keepalives, and the lack of
real-time guarantees in Linux’s scheduling policy.

Thesis Supervisor: Stephen J. Garland
Title: Principal Research Associate
Acknowledgments

This project was funded under a joint research agreement between MIT and Vanu, Inc. I owe a great deal of thanks to John Chapin and Vanu Bose for helping me turn this work into a Master’s thesis and giving me the opportunity to work on such a unique project. John, Vanu, and Andrew Beard ironed out all the intellectual property and financial issues to make this arrangement possible.

I also owe a great deal of thanks to my advisor, Stephen Garland, for playing an active role to make this arrangement work. During our weekly meetings, Stephen provided me with valuable feedback and raised important questions. He helped me stay focused, proofread all my drafts, and improved the quality of my writing immensely.

The infrastructure, which included everything from the signal processing libraries to the cluster work, was critical to this thesis. My thanks extend to all of the employees of Vanu, Inc. who contributed to various parts of it. Victor Lum, Andrew Chiu, Jeff Steinheider, and Alok Shah answered many of my signal processing questions and helped me get a failover demo working.

Much of my work would not have proceeded without Steve Muir, whom I worked with on a daily basis. Steve provided much needed assistance with the implementation of the failover system. He also added key features and fixed a number of bugs for me. His help saved me countless hours of trouble.

I would also like to thank my brother, Roger, for recruiting me to this project and working on the initial parts of the cluster.

Last but not least, I would like to thank my parents for their love and support.
Contents

1 Introduction ........................................... 13
   1.1 Overview ........................................ 13
   1.2 What is software radio? ......................... 14
   1.3 Motivation for using a distributed system .... 15
   1.4 Scope of research .............................. 16
   1.5 Organization of thesis ......................... 16

2 Failure Detection Requirements .................... 19
   2.1 Terminology ..................................... 19
   2.2 Causes of failures .............................. 19
   2.3 What failures should we detect? ................ 20
   2.4 What will NOT be detected ...................... 21
   2.5 Detection times ................................ 22
   2.6 Accuracy ....................................... 22
   2.7 Overhead ....................................... 23
   2.8 Multiple failures .............................. 23

3 Software radio cluster ............................. 25
   3.1 Gigabit network ................................ 26
   3.2 Sprockit ....................................... 26
   3.3 Radio Description Language (RDL) ............. 27
   3.4 Network streams ............................... 30
   3.5 CORBA ......................................... 32
4 Failover Design 35
4.1 Failure Detection 35
  4.1.1 Why keepalives? 35
  4.1.2 Monitoring structure 37
  4.1.3 Partial configuration based on connectivity 38
4.2 Failover 38
  4.2.1 Central controller and backups 38
  4.2.2 Failover Coordination 39
  4.2.3 Connectivity issues 40
  4.2.4 Multiple failures 40
  4.2.5 Byzantine failures 43

5 Performance Evaluation 45
5.1 Keepalive model 45
5.2 Keepalive model for 0 < k < 1 49
5.3 Measuring failure detection times 51
5.4 Keepalive scheduling 52
5.5 Expected vs. measured results 53
5.6 False positives and negatives 53
5.7 Causes of false positives 56
5.8 Detection times vs. accuracy 57
5.9 Detection under stress 58
5.10 Overhead 60

6 Related work 63

7 Conclusions and Future Work 65
  7.1 Future Work 66
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-1</td>
<td>A FM radio implemented as a set of Sprockit objects</td>
<td>28</td>
</tr>
<tr>
<td>3-2</td>
<td>Source code for specifying an assembly of an FM radio in RDL</td>
<td>29</td>
</tr>
<tr>
<td>3-3</td>
<td>Sample source code for an FM radio in RDL</td>
<td>31</td>
</tr>
<tr>
<td>3-4</td>
<td>A FM radio distributed across different machines</td>
<td>33</td>
</tr>
<tr>
<td>3-5</td>
<td>The application driver</td>
<td>34</td>
</tr>
<tr>
<td>4-1</td>
<td>A machine with modules that connects to multiple machines</td>
<td>41</td>
</tr>
<tr>
<td>4-2</td>
<td>A possible scenario where a machine and its neighbors fail</td>
<td>42</td>
</tr>
<tr>
<td>5-1</td>
<td>Keepalive model for $k \geq 1$</td>
<td>46</td>
</tr>
<tr>
<td>5-2</td>
<td>Keepalive model for $0 &lt; k &lt; 1$</td>
<td>49</td>
</tr>
<tr>
<td>5-3</td>
<td>Diagram of simulating failures and detecting times</td>
<td>51</td>
</tr>
<tr>
<td>5-4</td>
<td>Experimental results for failure detection times</td>
<td>54</td>
</tr>
<tr>
<td>5-5</td>
<td>Graphs of accuracy vs. detection times</td>
<td>59</td>
</tr>
<tr>
<td>5-6</td>
<td>False positives vs. detection times for loaded system</td>
<td>61</td>
</tr>
<tr>
<td>5-7</td>
<td>The CPU overhead of the detection system for one sender and receiver</td>
<td>62</td>
</tr>
</tbody>
</table>
List of Tables

5.1 Experimental results for failure detection times ............... 55
5.2 Experimental results for failure detection times under stress ...... 61
Chapter 1

Introduction

This thesis describes and evaluates a design for quickly detecting failures in distributed software radio applications. In the context of the work presented here, an application is composed of a number of connected signal processing modules that are distributed across multiple machines, and the focus is on determining how fast process crashes or network link outages can be detected for applications running on general-purpose processors and operating systems. We investigate how fast failures can be detected using standard application-level calls to exchange rapid keepalives between machines.

1.1 Overview

As new wireless standards emerge, the need for interoperability among different radio devices has grown. During the Gulf War, for example, the United States military struggled to coordinate its different armed service divisions because each used radio equipment that were incompatible with those of other divisions. In Grenada, the difficulty became especially apparent when officers had to use their personal calling cards to order air strikes [12]. On the commercial side, the dizzying array of competing cellular phone standards (e.g., AMPS, CDMA, TDMA, GSM) provides an example closer to home. Each standard uses its own frequency, power, and encoding methods, making it difficult for phones to interoperate. While some cellular phones offer both
analog and digital modes, none can support every standard.

Many of these devices, however, share something in common: they use digital signal processors (DSPs) to encode and decode information. DSPs are highly specialized for performing numerically-intensive computations. They provide instruction sets that optimize flow control and parallelize loading and processing data in real-time. However, in order to take advantage of the instruction sets, much of the software written for DSPs must be written by hand in low-level assembly language. As a result, DSP code has traditionally been highly specialized for one particular application. This specialization has made it hard for companies and people to keep pace with the ever-changing landscape of wireless standards. Adapting DSPs to use the latest, often incompatible, standards has been an expensive and time-consuming process.

1.2 What is software radio?

One version of software radio (also called virtual radios) is a technology where all the signal processing is implemented in software [7]. This approach uses general-purpose, commodity workstations instead of DSPs to perform signal processing in user space. The hope is that using commodity processors will reduce costs of custom-built hardware and alleviate many of the upgradability problems of DSPs by giving a device the flexibility to perform many types of functions. In theory, changing a device to use a different wireless standard would be a simple matter of running another application.

Software radio also offers other benefits. Bose described ways that using general-purpose processors might make spectrum use more efficient and flexible [6]. Physical layers could adapt to different environments and requirements. Unused portions of the spectrum could be allocated on-the-fly to meet users’ demands.

The availability of their rapidly-improving components, operating systems, and development tools make personal computers (PCs) an attractive software radio platform. PC components are produced in mass quantities and made readily available at lower prices than custom-built hardware. Linux, a freely available operating system, continues to include new features, support new hardware, and offer increasing stabil-
ity with every revision. A growing number of real-time applications are using Linux as the operating system of choice.

1.3 Motivation for using a distributed system

There are several reasons motivating the use of a distributed system for software radio applications:

Computational demands Although processing power continues to track Moore's Law, many software radio applications still demand far more resources than a single, state-of-the-art machine can provide. While it may be possible to perform a large amount of signal processing—ranging from computing a finite impulse response to convolution decoding of an input signal—a single Pentium processor is often inadequate. An AMPS cellular basestation, for example, must process over 20 voice and control channels simultaneously. Currently, a 700 MHz Pentium processor can only handle approximately 6 channels.

To meet these computational demands, multiple computers could be employed to form a distributed system. Using a distributed system connected by a local-area network offers the ability to distribute the workload, parallelize computations, and scale as the need arises.

Reliability Another reason for using a distributed system is to provide reliability. Commodity PCs provide an inherently unreliable environment for distributed software radio applications. Computers lose power, processes crash, operating systems hang, and network outages occur. Human errors and environmental factors compound the risk of a failure. Many signal processing applications, however, have stringent reliability requirements. For example, phone companies that operate cellular basestations demand over 99.999% availability, or 5.3 minutes of downtime a year. Downtime costs money and can even result in the loss of lives.

Current basestations use reliable DSPs and other components to achieve high availability. Moreover, the software that runs on top of these components can be
tested very rigorously because it has far fewer states than software that runs on commodity PCs. However, many fault-tolerant systems have been built with unreliable components by adding redundant parts and providing mechanisms to mask faults. With these techniques, clusters of commodity PCs could potentially achieve a high degree of availability.

1.4 Scope of research

To prevent a total system failure, a high-availability system must do two things: detect faults and recover from them. This paper will focus primarily on a design for detecting faults for a cluster of personal computers running software radio applications.

In other settings, many of the difficulties in recovering from a fault reside in deciding when and how to preserve state over certain intervals, or checkpointing. In our setting, many software radio applications have little state or infrequently-changing state. Decoding a color television signal, for example, requires very little state. Beyond the initial settings for the input and output rates, an application that decodes NTSC signals uses only the digitized samples on a frame-by-frame basis to find the horizontal and vertical syncs.

In applications that retain little state, a mechanism to detect failures quickly and accurately is a key step to recovery. Recovering from a failure will likely be faster than detecting the failure. One feature that help makes recovery fast is the use of hot backups. That is, backup machines will be configured to replicate the state of primary machines so that they can quickly take the places of failed machines.

1.5 Organization of thesis

Chapter 2 describes the requirements for the failover system. Chapter 3 details the infrastructure and implementation of the distributed software radio cluster. Chapter 4 discusses the design of the failover system for this cluster, and chapter 5 evaluates
this design through various models and experiments. Chapter 6 describes related work, and chapter 7 discusses the conclusions of this thesis and directions for future work.
Chapter 2

Failure Detection Requirements

2.1 Terminology

At the system level, a failure occurs when the application deviates from its correct behavior. A fault is the underlying defect of the system, such as a design flaw or programming bug. An error occurs as a result of the fault. For example, a fault could be a statement that divides by zero. However, if the code is never executed, it will never result in an error. Therefore, a fault does not cause any problems unless an error results from it.

Most of this paper will look at failures from the perspective of individual machines. From this perspective, when a machine crashes, it is deemed a failure. On the other hand, from the application perspective, the crashed machine is considered an error. The error may or may not lead to a failure in the application, depending on the application’s ability to recover from loss of a machine.

2.2 Causes of failures

What could cause an application to fail? Gray [18] found that many of the causes of system failures involved software or human errors, not hardware problems. As hardware has become increasingly reliable, software bugs and other errors have taken over as the dominant causes of failures. Errors can be categorized as arising from the
following sources of faults:

- Environmental (e.g., storms, fire, power outages)
- Hardware (e.g., disks, controllers, processors, cables, switches)
- Software (e.g., bugs, memory leaks, configuration errors)
- Operational (e.g., operating system limits, upgrades)

Making distributed software radio applications that are highly available is challenging due to the number of things that can go wrong. It is possible to reduce, but not eliminate, software problems with extensive testing and modular design. However, distributed systems do not operate in an isolated environment. They are subject to random events, such as natural disasters, and human actions that lead to failures. The overarching goal is to mitigate the risk by detecting errors as quickly and accurately as possible.

2.3 What failures should we detect?

To use a specific example, suppose we have a simple distributed application that simultaneously demodulates multiple FM radio stations. One machine might be responsible for sampling analog data from an antenna and broadcasting the digitized information to other machines in the cluster. Those machines would then be responsible for demodulating different ranges of frequencies.

In such a scenario, one definition of failure is the inability to produce some single demodulated signal within a certain time. The cause of this failure may be the result of any of the aforementioned errors. A malfunctioning power supply could cause the machine that samples analog data to crash entirely, precluding any of the other machines from doing any useful work. Alternatively, the code for demodulating one of the frequencies may cause a segmentation fault. Despite these potential problems, the end result is likely to be the same: either the process will have crashed or the
machine will have “disappeared” from the network. We will focus on detecting two failures:

**Process crashes** Since software is error-prone, a running process can crash due to bugs or problems in its code. A process might access an illegal memory address or get into an unrecoverable state. The problem may not manifest itself for a long time, as in the case of a tiny memory leak. However, a single process crash can cause a total system failure if each process is dependent on another to produce data.

**Network drop-offs** Even if the process is working properly, a machine could “disappear” from the network for many reasons. For example, the Ethernet cabling to the machine could be cut, the port for its switch could malfunction, or an operator might misconfigure the machine. In addition, operating system, power, and component failures could cause a machine to drop off the network. In any case, the actual cause of one failure may be indistinguishable from another, but the result is the same: the machine will not be reachable from the network.

### 2.4 What will NOT be detected

There are some failures that we will not attempt to detect. They include:

**Data validity** Correct behavior often means that the system produces an intended output given a specific input. Validating whether this output is correct, however, can be difficult because correctness depends on the application. Therefore, to avoid getting tied to one specific application, this failure detection mechanism will not verify the output but only attempt to ensure that it is produced.

**Byzantine failures** Byzantine failures occur when processors behave maliciously. For example, a machine might send messages that announce that every machine has failed. Byzantine failures require much more complex behavior because individual machines cannot be trusted, so some method of obtaining consensus
among multiple machines must be used. Stoller and Nett described voting mechanisms that deal with achieving consensus [25] [29].

**Link layer failures** Link layer failures occur when corrupted packets or invalid bits are sent across the network. If an application uses the UDP protocol to transmit data, dealing with these errors require some end-to-end verification schemes. These schemes add additional complexity and overhead. For this system, we will assume that link layer failures will not occur.

### 2.5 Detection times

The exact requirements for detection times depend largely on the specific application, but a typical upper bound requirement is to detect a failure within 50 milliseconds. To a human ear, a 50 millisecond gap during an audio stream will sound like a short click. While this gap will be noticeable, it is acceptable for most voice applications, which have many other sources of noise. This 50 millisecond requirement is an order of magnitude faster than for transactional database systems, which typically can tolerate failover times on the order of seconds.

### 2.6 Accuracy

Availability is computed in terms of Mean Time To Failure (MTTF) and Mean Time To Recover (MTTR) by the formula:

\[
\text{Availability} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}}
\]

Highly available systems attempt to maximize MTTF and minimize MTTR. Reducing detection time reduces MTTR and increases availability. Detection times could be decreased by using very aggressive timeouts, but this leaves less room for error and opens the way for more false positives. If a false positive occurs, the system will initiate recovery needlessly. Therefore, a high rate of false positives reduces the
MTTF because failures will be reported more often than they actually occur. For this reason, we wish to minimize false positives.

On the other hand, a failure that can be detected 100% of the time but only minutes after it happened has two problems. First, MTTR increases because it takes a relatively long time to detect a failure. Second, the time it takes to detect that failure falls short of the requirements. Therefore, we would like to find a good balance between the accuracy of the failure detection mechanism and detection times.

2.7 Overhead

The detection mechanism must be able to work transparently to the application and add minimal computational and network overhead to the system. It should still operate under heavy processing and network loads.

2.8 Multiple failures

Some systems make the assumption that multiple failures do not occur simultaneously. In practice, they do. For example, a power cord could be cut, causing all machines that were plugged into it to turn off. A failure detection mechanism for software radio applications should be able to cope with multiple failures.
Chapter 3

Software radio cluster

This section describes the cluster for which the fault detection mechanism will be built. Machines in the cluster run Linux on Pentium III processors. A software radio application, which consists of a set of connected signal processing modules, is constructed by writing a high-level description in the Radio Description Language (RDL). An application driver interprets the RDL program, connects modules, and sets parameters. Data flows from one module to a downstream module, forming a pipeline or graph.

Signal processing modules can be placed on different machines. By default, every module is placed on a single machine. However, an application writer can place modules on other machines. To make it easy to do this, the RDL compiler automatically handles the details of sending data from one machine to another. When it finds two connected modules that are located on different machines, it inserts special modules into the pipeline that send data from the upstream machine to the downstream machine.

Each machine in the cluster runs a process that awaits commands from the driver. This process has the ability to create, connect, and initialize any signal processing module available in its library. When an application runs, the driver contacts these processes via CORBA to configure machines as specified by the RDL code. Once all the machines have been initialized, the driver notifies all machines to begin working, and data proceeds to flow down the pipeline.
3.1 Gigabit network

To support applications that process high-bandwidth data streams (over 100 Mbit/s), the cluster of machines is connected on a local area network over a gigabit switch. Each machine can theoretically transmit up to 1000 Mbit/s to another machine in both directions without packet loss. Because the cluster is connected on a local area network, packet latencies are very low (on the order of microseconds). We will exploit this property to achieve fast failure detection.

Using nttcp [2], we measured the throughput between two machines on the gigabit Ethernet for different protocols. The first line of the output for each protocol shows the nttcp results for the sender, and the second line shows the results for the receiver. The results for UDP show a throughput of approximately 960 Mbit/s:

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Real s</th>
<th>CPU s</th>
<th>Real-MBit/s</th>
<th>CPU-MBit/s</th>
<th>Real-C/s</th>
<th>CPU-C/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>1409600000</td>
<td>3.33</td>
<td>0.96</td>
<td>960.8103</td>
<td>3333.3333</td>
<td>30026.22</td>
<td>104169.8</td>
</tr>
<tr>
<td>1409575424</td>
<td>3.33</td>
<td>2.58</td>
<td>960.5375</td>
<td>1240.2357</td>
<td>30017.10</td>
<td>38757.8</td>
</tr>
</tbody>
</table>

TCP, however, shows a throughput of approximately 229 Mbit/s:

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Real s</th>
<th>CPU s</th>
<th>Real-MBit/s</th>
<th>CPU-MBit/s</th>
<th>Real-C/s</th>
<th>CPU-C/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>1409600000</td>
<td>13.98</td>
<td>0.97</td>
<td>228.9306</td>
<td>3298.9691</td>
<td>7154.08</td>
<td>103092.8</td>
</tr>
<tr>
<td>1409600000</td>
<td>13.98</td>
<td>1.67</td>
<td>228.9144</td>
<td>1916.1677</td>
<td>9823.86</td>
<td>82232.3</td>
</tr>
</tbody>
</table>

The difference between UDP and TCP’s throughputs has important implications for software radio applications. If the application requires TCP’s flow control and reliability guarantees, then the maximum bandwidth it can use is only a quarter of that available to UDP.

3.2 Sprockit

Signal processing modules are implemented as C++ objects in a framework called Sprockit. Example modules include those that perform FM demodulation or find the NTSC television horizontal syncs. Each module falls into three different categories:
sources, sinks, and filters. A source gathers data and sends it down the pipeline. A filter takes data, performs some processing on it, and passes it down the pipeline. A sink receives data and forms the end of the pipeline.

The simplest application can be constructed by taking a source and connecting it to a sink. More complex applications can be made by adding filters. Figure 3-1 shows an FM radio application constructed with Sprockit. The first item, “Echotek A/D Source,” is a software module that retrieves data from an Analog to Digital (A/D) converter. The A/D converter samples a 5 MHz band from an Radio Frequency (RF) frontend, and the module passes this data down the pipeline. The frequency translating filter then selects a 200 KHz channel from the band, centered around the desired station (e.g., 107.9 MHz). The channel then gets demodulated. A low pass filter then strips out extraneous data, leaving the mono and stereo FM signal intact. At the last stage of the pipeline, the signal is played through a sound card.

Applications can be coded in C++ with Sprockit. However, maintaining code for applications that have many modules can be cumbersome. Groups of modules form processing chains that are often duplicated, which leads to copied code. In addition, to make a distributed application, an application writer must break up the data pipeline and manually insert special modules to send data across the network.

3.3 Radio Description Language (RDL)

To make it easier to create a distributed software radio application, the Radio Description Language (RDL) was created. RDL allows a programmer to specify which modules should be instantiated, how they should be connected, and where they should reside. Figure 3-2 shows sample RDL code for the FM radio application. First, the basic set of modules are grouped into an assembly. The connections between each modules are specified by arrows (e.g., echotek -> chanSelect -> fm).

Once the assembly is defined, the application writer can choose to place modules on different machines. Figure 3-3 shows the Java code that is used to do this. The main() function instantiates the actual application and designates that all modules
Figure 3-1: A FM radio implemented as a set of Sprockit objects
// the FmRadio assembly
package tests;

include "vanuinc/UnifiedEchotekSource.rdl";
include "vanuinc/FreqXlatingLpf.rdl";
include "vanuinc/FmDemod.rdl";
include "vanuinc/LowPassFilter.rdl";
include "vanuinc/OssDspSink.rdl";
include "vanuinc/Wj8655Controller.rdl";

assembly FmRadio
{
    // RF frontend controller module
    module Wj8655Controller wjController;

    module UnifiedEchotekSource echotek;
    module FreqXlatingLpf chanSelect;
    module FmDemod fm;
    module LowPassFilter lpf;
    module OssDspSink speaker;

    echotek -> chanSelect -> fm;
    fm -> lpf -> speaker;
}

Figure 3-2: Source code for specifying an assembly of an FM radio in RDL
by default belong on machine “bandy” (lines 9-12). However, the low pass filter and
the sound card modules are assigned to machines “giants” (lines 14-16) and “niners”
(lines 18-20). At the bottom of Figure 3-3, the FmRadio_echotek class inherits the
assembly class, which defines the application’s modules based on the file in Figure 3-2.
The FmRadio_echotek class is responsible for setting the parameters of the modules,
such as the sampling rate for the A/D converter.

3.4 Network streams

If an application specifies that modules should reside on different machines, the
RDL compiler automatically handles the details of inserting modules that will send
and receive data across the network. The compiler does this by inserting two modules,
a NetSender and a NetReceiver, into the pipeline. Suppose there are two machines,
A and B, specified in an RDL application. In this case, a NetSender is added to the
end of the pipeline of machine A, and a NetReceiver is added to the beginning of the
pipeline of machine B. From the perspective of a single machine, a NetSender is a
sink because it is machine A’s last module. The NetReceiver, on the other hand, is a
source because it begins the flow of data on machine B.

The NetSender receives data from its upstream modules and sends it across the
network via a socket to a NetReceiver. The application can use either UDP or TCP,
depending on its requirements. Upon request from its downstream modules, the
NetReceiver passes the data that it received from the NetSender.

Figure 3-4 shows a figure of a version of the FM radio distributed across three
machines. Each gray box encapsulates modules on one machine. The NetSender
and NetReceiver modules have been inserted to send data from one machine to an-
other. The first machine has modules that access the A/D converter (“echotek”) and
configure the RF frontend (“wjController”). The output of the FM demodulator
module (“fm”) goes to a NetSender (“streams1.netSender”), where it is sent across
the network to the NetReceiver on the other machine (“streams1.netReceiver”). The
pipeline continues until the final output is played by the sound card in the third
public class FmRadio implements RdlMain
{
    public void main(DriverArgs args)
    {
        FmRadio_echotek radio = new FmRadio_echotek(Rdl.root());

        ArrayList c1 = new ArrayList(1);
        c1.add("");
        c1.set(0, "bandy");

        Machine m1 = Machine.Get(c1);
        radio.setMachine(m1);

        c1.set(0, "giants");
        Machine m2 = Machine.Get(c1);
        radio.lpf.setMachine(m2);

        c1.set(0, "niners");
        Machine m3 = Machine.Get(c1);
        radio.speaker.setMachine(m3);

        Rdl.install(radio);
    }
}

public class FmRadio_echotek extends FmRadioAss
{
    public FmRadio_echotek(Assembly parent)
    {
        super(parent);
    }

    public void pfunction()
    {
        // Parameters set here
        .
        .
        .
    }
}

Figure 3-3: Sample source code for an FM radio in RDL
machine ("speaker").

3.5 CORBA

CORBA is used in a limited fashion to make Remote Procedure Calls (RPCs) to machines in the cluster. Each machine in the cluster runs a process that registers itself with the application driver upon startup. As soon as all the machines are ready, the driver invokes methods on each machine to create, connect, and initialize modules.

It is important to note that CORBA is not used to pass the actual data used in the signal processing from one machine to another. CORBA is not suited for sending mass amounts of data over the network, nor can it be used for detecting a fault quickly. CORBA invocations timeout on the order of seconds, not milliseconds.

3.6 The application driver

Figure 3-5 shows how the application driver operates. First, the driver takes the RDL application and compiles it into its internal representation, inserting NetSender and NetReceiver modules where they are needed. Once the application is compiled, the driver invokes CORBA methods on remote machines (e.g., A, B, and C) to create, connect, and initialize modules. Once this is done, the driver notifies all the machines to begin working.
Figure 3-4: A FM radio distributed across different machines
Figure 3-5: The application driver
Chapter 4

Failover Design

4.1 Failure Detection

The design of the failure detection mechanism revolves around the use of rapid heartbeats, or keepalives. Each machine in the cluster sends keepalives to its connected neighbors to indicate that it is still operating. If no keepalives have been received for some time, a machine will be considered dead. Exactly how often keepalives should be sent, how often they are checked, and how many can be missed will be evaluated to see how detection times are affected.

The focus of this work is on measuring how quickly and accurately a failure can be detected under a number of conditions. First, each machine is running a general-purpose processor with a standard operating system that makes no real-time guarantees. Second, each processor is loaded by signal processing computations while monitoring for failures. Lastly, the software radio application is the primary process running on the system.

4.1.1 Why keepalives?

Keepalives have been commonly used in distributed systems to detect failures. The SunSCALR System, for example, uses IP multicast to send keepalives to every machine in the cluster [27]. However, because it uses relatively conservative
timeouts—on the order of seconds—to detect failures, multicasting keepalives is a low-bandwidth operation. To meet the requirements for software radio applications, each machine in this cluster will send keepalives much more frequently and use faster timeouts.

Sending a keepalive is an indication of liveness, a property that something "good" will eventually happen [15]. A machine that is able to send keepalives shows that its process is running and its network interface is operating properly, assuming that Byzantine failures do not occur. Keepalives also have other properties that make them desirable:

**Flexibility** Since failures must be detected within some time dictated by specific applications, a detection mechanism must be able to adapt to different requirements. On one extreme, some applications, such as telnet, may not need stringent detection times. These applications could use an option in the TCP protocol that enable the sending of a keepalive if no activity has occurred within two hours [28]. On the opposite end, software radio applications need detection times on the order of milliseconds or tens of milliseconds. The ability to configure the frequency of sending and receiving keepalives provides flexibility to meet different requirements.

**Failure isolation** Keepalives also provide a way of isolating a failure to a particular machine. Since each machine can send keepalives independently of its neighbors, it is possible to identify exactly which machine failed. On the other hand, techniques that rely on checking the flow of data through the processing pipeline do not have this property. A stall in the pipeline could be caused by any upstream machine, not just an immediate neighbor.

**Minimal overhead** While sending keepalives at a fast rate will facilitate fast detection, it adds additional network and computation overhead to the system. However, the overhead is minimal. A machine sending a 32-byte packet every millisecond generates 0.256 megabits/s. At that rate, 100 machines sending keepalives to one machine generates about 26 megabits/s. This leaves enough
bandwidth for normal traffic within the gigabit network. In addition, the overhead for computational is minimal as well. It requires waking up at certain intervals, performing some reads and writes, and determining whether a failure has occurred.

4.1.2 Monitoring structure

There are several possible ways to structure how machines monitor other machines for failures:

**Star** In the star configuration, a single machine monitors all other machines. All machines in the cluster would send keepalives to this monitor. The advantage of this configuration is that it is simple to design and easy to implement. However, this configuration creates a single point of failure: if the monitoring machine fails, then all subsequent failures will not be detected.

**Full** Instead of having a single machine monitor for failures, the full configuration has each machine monitor all other machines, which broadcast or multicast keepalives. This allows every machine to detect almost any failure that occurred. The advantage is added redundancy. If three out of four machines fail, for example, then the remaining machine will be able to detect each of the failures independently. The disadvantage to this approach is that it does not scale. As more machines are added, the number of connections grows quadratically ($O(n^2)$).

**Partial** Another approach is to have each machine monitor only a subset of other machines. This solves the scalability problems in the full configuration at the cost of reducing redundant monitors. By limiting the number of connections between machines, the system can benefit from some redundancy, albeit not as much as it would have with the full configuration.
4.1.3 Partial configuration based on connectivity

The partial configuration represents the best tradeoff. The single point of failure created by the star configuration is very unattractive; depending on a sole machine makes the system less reliable. The lack of scalability in the full configuration limits its usefulness. To achieve fast detection times, it may be necessary to send keepalives rapidly. As the frequency increases and the number of nodes grows, sending keepalives to every machine may produce too much network traffic and processing overhead.

Machines that have modules that send data from one machine to another are said to be composed. Because an RDL application is represented as a directed graph of signal processing modules, the connectivity of the machines in the cluster can also be expressed as a directed graph, with each node having incoming and outgoing arcs.

Our system uses the partial configuration by having connected machines monitor each other. Each machine sends keepalives to each of its outgoing and incoming neighbors. Neighbors therefore monitor their neighbors.

4.2 Failover

A basic failover mechanism will be used in the system. The failover mechanism will be concerned with connecting backup machines into the processing pipeline. It will not, however, deal with more complex behaviors, such as data stream resynchronization or message replaying.

4.2.1 Central controller and backups

A fundamental decision that impacts the failure detection design was to use a very reliable, central controller that coordinates a number of tasks, including failover and load balancing. The RDL application will be launched from the controller machine as well.

To minimize the dependency on the controller, the detection mechanism will only rely on the controller to allocate and configure hot backups that can take over for
machines that have crashed. Each machine in the cluster will be either a primary or a backup machine. A primary machine is the one that is currently active in the pipeline. A backup machine has identical modules and settings that are allocated on a primary machine. If a primary machine fails, the backup will take over, and the controller will allocate a new backup machine to take its place.

The use of hot backups is motivated by several reasons. First, it speeds failover times because a backup can move quickly into action as soon as a primary machine has failed. Second, hardware requirements restrict the pool of available backup machines. For example, a primary machine may need to communicate with a RF frontend through a serial port, and it may only be practical to wire a few, not all, machines to the frontend. Similarly, only a few backup machines may have A/D converters that are needed to start the data pipeline.

For the sake of design simplicity, we assume that there will be one hot backup for each machine running at a time. Every machine therefore has exactly one corresponding backup. This assumption will become important when the system must decide which backup should take over for a failed machine.

In the future, the system will optimize the number of backups that are needed. To conserve resources, it would be desirable to have backup machines serve multiple primary machines. It would also be desirable to have multiple backups for one machine. However, both these optimizations require additional coordination if a failure occurs. The system must somehow decide which backup machine should become active and ensure that only one does.

4.2.2 Failover Coordination

Every machine in the cluster can send and receive on a shared multicast channel for control messages. This channel is separate from the ones used for keepalives. Control messages are used to facilitate failure diagnosis and recovery. The possible messages that be can sent are:

- Is Machine X alive?
- Machine X is dead
- Machine X is alive
- Machine Y has taken over for machine X

Backup machines monitor the control channel for reports on failures on primary machines. When a primary machine X fails, the monitoring machines announces, “Machine X is dead.” Upon hearing this message, the corresponding backup machine Y announces, “Machine Y has taken over for machine X.” All machines that were connected to X will now connect to Y. Since a backup machine is responsible for declaring this switch, the switch happens once and only once. Multiple machines can report that machine X has failed within a small time, but none will try to connect to a backup until a switch message is announced.

If Y does not announce a switch message after a pre-determined number of “Machine X is dead” messages, the controller sends a query, “Is Machine Y alive?” If no response is heard within some time, the controller creates a new backup for X to take Y’s place.

4.2.3 Connectivity issues

Because machines can have modules that are connected to different machines, the failover system must be able to connect the right modules to the right machines. As shown in Figure 4-1, machine A can send data to machines B and C. If B fails, then the modules on A must know how to connect to B’s backup while retaining the connections to C. To deal with this issue, each machine creates a table that indexes modules to their connected machines. When a machine switch is announced, this table is used to determine which modules should connect to the new machine.

4.2.4 Multiple failures

Multiple failures are also handled in a systematic way. If multiple failures occur, it is possible that a graph will be partitioned in such a way that a machine may be
Figure 4-1: A machine with modules that connects to multiple machines
left without any connected neighbors. Consider Figure 4-2. If B, C, and D fail, C will have no neighbors that can detect that it has failed as well. We call C a disconnected node.

![Figure 4-2: A possible scenario where a machine and its neighbors fail](image)

To deal with disconnected nodes, we assume that at least one machine will remain up when a failure occurs. Since the graph of machines is connected, we know that at least one backup will take over for a failed machine. Once this occurs, this backup will monitor its neighbors to determine whether any have failed. If any of them have, then another backup connects into the pipeline. In this way, disconnected nodes will be restored soon after their backups become active.

The advantage of this scheme is that it makes few assumptions about which ma-
chines must stay up when multiple failures occur. It makes no assumptions about which machines must remain up; it only assumes that one does. A different approach would be to assume that at least one monitor remains up for each node. This assumption guarantees that disconnected nodes never occur. In Figure 4-2, for example, we would have to presume that either B or D must stay up to detect a failure in C. However, this is a poor assumption because it is impossible to predict which machines will fail.

However, the disadvantage of waiting until backups are restored is that recovery time increases linearly as a function of the number of disconnected nodes. In the future, we could mitigate the chance that a machine will become a disconnected node by adding redundant monitors.

4.2.5 Byzantine failures

Byzantine failures pose significant problems for software radio applications. First, Byzantine failures make failure detection more difficult. If machines can impersonate others or send erroneous messages, then no assumptions can be made about the validity of reports. A machine that repeatedly reports that other machines have failed will cause recovery to be initiated needlessly. In addition, because modules must be connected properly for the application to produce the intended output, recovery also rests on valid switch messages being sent.

To protect against these kinds of Byzantine failures, some cryptographic techniques for message authentication and verification could be used. Machines would then need to provide some sort of proof of identity before a control message is accepted. Having this kind of protection would be a good safeguard to have if impostors become a significant problem in the future. However, the complexities of creating, managing, and using cryptographic keys may outweigh the benefits of using them.

Second, if Byzantine failures can occur, any machine performing signal processing computations would be suspect. Castro and Liskov described one practical way of implementing a Byzantine protection mechanism that multicasts a request to multiple machines, tallies the responses, and takes the majority vote as the final result [9].
These mechanisms do not make sense for software radio applications because they constantly check the results of deterministic function calls. In software radio applications, most of the computational work is performed on high-bandwidth, streaming data. Continuously checking the computational results from multiple machines would degrade performance in an unacceptable manner.
Chapter 5

Performance Evaluation

In this chapter, we evaluate the speed and accuracy with which failures can be detected. We ran a number of experiments, varying the sender and receive intervals and the thresholds for the number of missed keepalives. To provide a basis for comparison, we first present a model of the keepalive mechanism.

5.1 Keepalive model

Figure 5-1 is a model for the keepalive mechanism. Each hash mark on the top line shows when the sender writes a keepalive to the network. Each hash mark on the bottom line shows when the receiver checks for keepalives. We assume that network transit times, along with any time spent reading and writing keepalives, are negligible. We also assume that a machine ceases to send keepalives as soon as it fails.

In this model, the expected time to detect a failure is influenced by several variables. First, it is affected by $n$, the number of keepalives that can be missed in a row before a failure is signaled. $s$ is how often the machine sends a keepalive. For now, $k$ is the integral number of regularly-spaced keepalives that are sent between the times the receiver checks for them. Therefore, $r$, the interval between successive times the receiver checks for keepalives, equals $ks$. The sender and receiver intervals are offset by some phase difference $\Phi$, which can vary from one execution to the next if the receiver begins some random time after the sender.
The range of when failure can be detected is at $3r$. The number of keepalives missed in a row before a failure is signaled is $n$. $s$, $2s$, and $3s$ represent the sender intervals. $r$, $2r$, and $3r$ represent the receiver intervals.

$k = 2$

$n = 1$

$Y$

$n = 2$

$X$

$n = 3$

Sender

Receiver

$n = $ number of keepalives missed in a row before a failure is signaled
$s = $ sender interval
$r = ks = $ receiver interval
$\Phi =$ phase difference between sender and receiver

Figure 5-1: Keepalive model for $k \geq 1$
\( \Phi \) is not well-defined if \( k \) is not an integer. If \( k \) is not an integer, then the number of keepalives sent during each receive interval will change. Thus, there could be multiple or infinite number of values for \( \Phi \). This would change the expected value of \( \Phi, E(\Phi) \). To keep the model simple, we restrict our attention to integral values of \( k \).

The time of detection, \( D \), is the difference between the time at which a failure is detected and the time, \( F \), at which the failure actually occurred. Assuming \( F \) is uniformly distributed over the interval between successive keepalives, the expected value of \( D \) is:

\[
E(D) = \left( \frac{L(D) + U(D)}{2} \right)
\]  

(5.1)

where \( L(D) = \) lower bound of \( D \) and \( U(D) = \) upper bound of \( D \)

As shown in Figure 5-1, it takes at least time \( \Phi \) to detect one missed keepalive, assuming the failure occurs some infinitesimally small amount of time before the sender is expected to send that keepalive. However, because the receiver must wait for \( n \) missed keepalives before signaling a failure, it must wait an additional \((n - 1)s\). Therefore, the lower bound \( L(D) \) is:

\[
L(D) = E(\Phi) + (n - 1)s
\]

(5.2)

\( L(D) \) is the fastest detection time that we can possibly obtain. A failure will be detected in time \( L(D) \) if it occurs as late as possible, or right before a keepalive should have arrived. However, failures can occur earlier and still be detected in the same receive interval. Therefore, the upper bound \( U(D) \) is equal to \( L(D) \) plus some additional time.

How much is this time? As we have just seen, a machine can fail at time \( X \) right before a keepalive should have been sent. Now we wish to find how early a failure can occur so that the receiver will detect it at the same time as a failure at \( X \). If we shift
X to the left by an entire receive interval time, \(ks\), then the failure will be detected during the previous receive interval. Therefore, for the failure to be detected during the same, not previous, interval, it must happen an infinitesimally small time after \(X - ks\). Thus:

\[
U(D) = L(D) + ks
\]  

(5.3)

As an example, consider X and Y for \(n = 2\) in figure 5-1. A failure can occur as late as \(X\) for the receiver to detect it at \(3r\), indicated by the vertical, dotted line. The distance between \(X\) and \(3r\) is \(\Phi + s\). For a failure to be detected at \(3r\), the earliest point at which it can occur is \(Y\). If a failure occurs before \(Y\), then the failure will be detected at \(2r\) instead of \(3r\).

Combining equations 5.1, 5.2, and 5.3 obtains:

\[
E(D) = \frac{2L(D) + ks}{2} = L(D) + \frac{ks}{2} = E(\Phi) + (n - 1)s + \frac{ks}{2} = E(\Phi) + (n + \frac{k}{2} - 1)s
\]  

(5.4)

\(n \geq 1\)

Assuming \(\Phi\) is uniformly distributed, \(E(\Phi)\) is \(\frac{s}{2}\). Equation 5.4 reduces to:

\[
E(D) = (n + \frac{k}{2} - \frac{1}{2})s
\]  

(5.5)

Equation 5.5 shows that if we are concerned solely with minimizing the detection time, we should try to minimize \(s\) because it has a multiplicative effect. The equation also confirms that increasing \(k\), which increases the receiver’s interval time, adds to
the detection time. For example, suppose \( n = 1, k = 1, \) and \( s = 10 \text{ ms} \). In this case, where a keepalive is sent every 10 milliseconds, we can expect to detect a failure in \( 10 \text{ ms} (\frac{1}{2} + \frac{1}{2}) = 10 \text{ ms} \). But if \( k = 2 \), then \( E(D) = \frac{3s}{2} = 15 \text{ ms} \). Similarly, increasing \( n \) also means that the receiver must wait longer before it signals a failure, which increases detection time as well.

### 5.2 Keepalive model for \( 0 < k < 1 \)

The previous model dealt with \( k > 1 \). When \( k > 1 \), the receiver knows that there should be at least one keepalive waiting every time it goes to check. If \( k \) is fractional, the receiver checks for keepalives more often than they are sent. That means \( n \) must be at least \( \frac{1}{k} \), lest the receiver always signal a failure.

![Diagram of keepalive model for \( 0 < k < 1 \)](image)

- \( n \) = number of keepalives missed in a row before a failure is signaled
- \( s \) = sender interval
- \( r = ks \) = receiver interval
- \( \Phi \) = phase difference between sender and receiver

If \( k \) is fractional, then there will be some receive intervals that will always “miss” a keepalive. Therefore, the receive intervals that follow will automatically incur a
number of missed keepalives.

Figure 5-2 shows a model for \( k = \frac{1}{2} \). At interval 5r, the receiver incurred a missed keepalive from interval 4r. If \( n = 2 \), then the receiver just needs to miss the keepalive that was sent right before 5r. Figure 5-2 shows that the latest point \( X \) at which a failure can be detected is right before 2s. The earliest point \( Y \) extends to the left by the width of the sender interval, as in the \( k \geq 1 \) model.

If we shift that entire range over by a sender interval, we see that this is the range where a failure can be detected at 5r for \( n = 4 \). If \( n = 4 \), a failure that occurs right before s will be detected at 5r because the 3r interval has already incurred one missed keepalive. Note that if \( n = 3 \), the failure will be detected at 4r due to the effect of incurred keepalives.

All this suggests that \( L(D) \), the lower bound on detection time, must take \( k \) and \( n \) into account. Since \( k \) is fractional, the receiver must wait for a certain number of receive intervals where no keepalives are read before a failure is signaled. Thus, \( L(D) \) is a function of \( r \). The number of intervals is \( n \) minus the number of incurred keepalives per interval, or \( \frac{1}{k} \). Therefore, \( L(D) \) is:

\[
L(D) = E(\Phi) + (n - \frac{1}{k})r
\]

\[
= E(\Phi) + (n - \frac{1}{k})ks
\]

\[
= E(\Phi) + (nk - 1)s
\]  \( (5.6) \)

\( U(D) \) is the same as before. Thus, \( E(D) \) is:

\[
E(D) = E(\Phi) + (nk - 1)s + \frac{ks}{2}
\]

\[
= E(\Phi) + (nk - 1 + \frac{k}{2})s
\]  \( (5.7) \)

\[
= (nk + \frac{k}{2} - \frac{1}{2})s
\]

\[
0 < k < 1, n \geq \frac{1}{k}
\]
Equation 5.8 is almost identical to Equation 5.5, except there is an extra factor of $k$ in the first term. Since $k$ is fractional and $n \geq \frac{1}{k}$, $nk$ must be at least 1. However, as $n$ increases, detection time increases slower than it would in Equation 5.5 because of the $k$ scaling factor.

5.3 Measuring failure detection times

Figure 5-3: Diagram of simulating failures and detecting times

To evaluate the performance of the detection mechanism, a number of experiments were run. Failure detection times are measured by shutting down a process and seeing how long the system takes to detect the death of the process. Before the process shuts down, it sends a message to the controller to log the current time. Once another machine detects that the process has died, it too sends a message to the controller. The controller records the difference in the arrival times of the two messages as the detection time. Figure 5-3 illustrates the process.

Another way to measure detection times would be to have machines A and B send
timestamps to the controller. Detection time would then be computed by subtracting the difference between the two timestamps. This way of computing detection time requires that both machines have their wall clock times synchronized with millisecond accuracy. This could be done using the Network Time Protocol (ntp). However, even with ntp, we had a difficult time making sure that every machine in the cluster was synchronized at all times. Instead, it was easier to use a single machine as a stopwatch.

5.4 Keepalive scheduling

The sending and receiving of keepalives are scheduled using the setitimer() system call. At regular intervals, setitimer() raises a signal, which is handled by a function call that manages the keepalives.

It is important that the signal handler be registered with the SA_RESTART flag. If it is not, any system call that is blocked when the signal is raised will return with the EINTR value. Some modules might interpret EINTR return value as a fatal error. Setting the SA_RESTART flag ensures that the function call is automatically restarted after the signal handler has completed.

Using setitimer() ensures that the application will temporarily break out of its signal processing loop to perform failure detection. An alternative way would be to use threads that wake up at certain intervals. This approach, which was tried initially, does not work as well as the setitimer() method. Because the keepalive thread has to compete with the main thread for processing time, there is a chance that it will be woken up a long time after the desired interval. The setitimer() is not completely immune to this effect either, but it will at least interrupt the signal processing loop when the process is running. In addition, threads incur some additional overhead with context switching.

Because the Linux kernel wakes every 10 ms (the jiffy interval), the minimum send and receive intervals cannot be set any faster than 10 ms. Therefore, if we wish to reduce $s$ below 10 ms, a lower-level mechanism must be used to trigger the handler. One way would be to program the local Advanced Programmable Interrupt
Controller (APIC) on a Pentium processor to generate interrupts at faster intervals. On multiprocessor machines, the APIC is enabled by default. However, because the APIC cannot be enabled on a uniprocessor machine without significant patches to the Linux kernel, this evaluation will deal only with intervals of 10 ms or more.

5.5 Expected vs. measured results

Figure 5-4 shows the results of several experiments used to measure detection times. The application was a test program constructed in RDL that continuously sent 40 Mbit/s of data from one machine to another. The graphs show that the experimental detection times approximately matched the expected detection times. From Table 5-4, we see that for most of the intervals the difference between the expected and measured time all fell within 6 ms of each other. For the entries where $k = 0.5$, the experiments confirmed that setting $n = 1$ generates 100% false positive rates.

5.6 False positives and negatives

In the experimental data, false positives registered negative detection times. When a false positive occurs, machine B reports that machine A has failed before it actually has. Therefore, step 3 occurs before step 2 in Figure 5-3. As a result, $t2 - t1$ is negative.

In theory, false positives are much more likely to occur than false negatives because monitoring machines check for failures very frequently. A false negative occurs when a machine does not detect a failure. However, in these experiments, false negatives never happened because that would require $t2$ to occur an infinite amount of time after $t1$. 
Figure 5-4: Experimental results for failure detection times. Each black bar represents the expected detection times computed from the keepalive model. Each white bar represents the mean detection time for 100 experiments over a 30 second interval.

\[ s = \text{sender interval} \]
\[ r = ks = \text{receiver interval} \]
\[ n = \text{number of keepalives missed in a row before a failure is signaled} \]
<table>
<thead>
<tr>
<th>s (ms)</th>
<th>r (ms)</th>
<th>n</th>
<th>expected (ms)</th>
<th>measured (ms)</th>
<th>difference (ms)</th>
<th>false positives (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td>1</td>
<td>10</td>
<td>13.9</td>
<td>3.9</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>2</td>
<td>20</td>
<td>24.4</td>
<td>4.4</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>3</td>
<td>30</td>
<td>34.2</td>
<td>4.2</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>4</td>
<td>40</td>
<td>43.5</td>
<td>3.5</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>5</td>
<td>50</td>
<td>53.4</td>
<td>3.4</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>20</td>
<td>1</td>
<td>15</td>
<td>20.2</td>
<td>3.8</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>20</td>
<td>2</td>
<td>25</td>
<td>27.8</td>
<td>2.8</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>20</td>
<td>3</td>
<td>35</td>
<td>38.9</td>
<td>3.9</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>20</td>
<td>4</td>
<td>45</td>
<td>48.8</td>
<td>3.8</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>40</td>
<td>1</td>
<td>25</td>
<td>28.7</td>
<td>3.7</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>40</td>
<td>2</td>
<td>35</td>
<td>37.5</td>
<td>2.5</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>40</td>
<td>3</td>
<td>45</td>
<td>46.4</td>
<td>1.4</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>40</td>
<td>4</td>
<td>55</td>
<td>59.5</td>
<td>4.5</td>
<td>3</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>1</td>
<td>20</td>
<td>22.7</td>
<td>2.7</td>
<td>3</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>2</td>
<td>40</td>
<td>43.7</td>
<td>3.7</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>3</td>
<td>60</td>
<td>61.8</td>
<td>1.8</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>4</td>
<td>80</td>
<td>82.5</td>
<td>2.5</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>5</td>
<td>100</td>
<td>103</td>
<td>3.0</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>10</td>
<td>1</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>100</td>
</tr>
<tr>
<td>20</td>
<td>10</td>
<td>2</td>
<td>15</td>
<td>19.6</td>
<td>4.6</td>
<td>4</td>
</tr>
<tr>
<td>20</td>
<td>10</td>
<td>3</td>
<td>25</td>
<td>28.8</td>
<td>3.8</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>10</td>
<td>4</td>
<td>35</td>
<td>38.6</td>
<td>3.6</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>10</td>
<td>5</td>
<td>45</td>
<td>48.8</td>
<td>3.8</td>
<td>0</td>
</tr>
<tr>
<td>40</td>
<td>20</td>
<td>1</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>100</td>
</tr>
<tr>
<td>40</td>
<td>20</td>
<td>2</td>
<td>30</td>
<td>29.6</td>
<td>0.4</td>
<td>3</td>
</tr>
<tr>
<td>40</td>
<td>20</td>
<td>3</td>
<td>50</td>
<td>53.4</td>
<td>3.4</td>
<td>0</td>
</tr>
<tr>
<td>40</td>
<td>20</td>
<td>4</td>
<td>70</td>
<td>76.2</td>
<td>6.2</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.1: Experimental results for failure detection times

\[ s = \text{sender interval} \]
\[ r = ks = \text{receiver interval} \]
\[ n = \text{number of keepalives missed in a row before a failure is signaled} \]
5.7 Causes of false positives

Processes can be pre-empted by the Linux kernel, and the operating system makes no real-time guarantees. Hence, the signals generated by the \texttt{setitimer()} calls may not be executed at the precise, intended intervals. The lack of real-time guarantees requires that detection times be increased to take deviations into account. Otherwise, false positives will result. They are often caused by:

- Competing processes or threads
- Skew in phase difference between sender and receiver ($\Phi$)
- Network packet delays

In theory, the first item should not be an issue because each software radio process is the only program running on the system. However, in reality, many services, like the Secure Shell daemon (\textit{sshd}), could be running in the background and become active on occasion. In addition, a hardware interrupt could take a few milliseconds to process in kernel space. Any slight interruption could delay the sending of keepalives, which would generate a failure report. While these interruptions technically could be considered a legitimate failure, we consider them false positives. Therefore, setting $n$ higher lowers the chance that these kinds of false positives from occurring.

Looking at network trace data, we can see how a skew in $\Phi$ can generate false positives. The following is an example trace from an experiment using $s = 10$ ms, $r = 10$ ms, and $n = 1$:

```
...  
10:56:33.166965 springbox.4151 > kiwis.2640: udp 4  
10:56:33.176965 springbox.4151 > kiwis.2640: udp 4  
10:56:33.186966 springbox.4151 > kiwis.2640: udp 4  
10:56:33.197606 springbox.4151 > kiwis.2640: udp 4  
10:56:33.197856 kiwis.2639 > 224.9.1.6307: udp 20 [ttl 1]
```

The first field shows the time with microsecond precision when a network packet arrived. The next field shows the source hostname and port from which the packet
originated. The destination host and port follows, and the remaining fields show miscellaneous information about the packet, such as its protocol and size.

The first four lines show machine springbox sending to kiwis every 10 ms. On the fourth keepalive, it arrives 11 ms afterwards. This slight deviation is enough to signal a failure, and we see that kiwis multicasts a failure report (224.x.x.x are multicast addressees). $\Phi$, the phase difference between the sender and receiver, has become small enough such that kiwis is reading packets less than 1 ms after they have been sent.

Lastly, network packet delays also could have a similar effect. On a local gigabit network, none of the experiments experienced significant packet loss or delays. However, if the network is working at full capacity, it is possible that the keepalives will experience delays or packet loss. We could give deal with this in several ways: increase the number of missed keepalives by raising $n$, reduce keepalive traffic by increasing $s$, or reduce the overall amount of data used in main pipeline. We could also reduce the receive interval by decreasing $k$, which has a similar effect as increasing $n$, but at less of a cost to detection time.

5.8 Detection times vs. accuracy

Several experiments were run to see the tradeoff between detection times and accuracy. In Figure 5-5a, the sender interval $s$ was set to the lowest possible value, 10 ms. Different values of $n$, the number of keepalives missed, were used to produce each point on the graph.

Each data point represents the results of running a simple application 100 times. The application consisted of a source that generated 40 Mbit/s worth of samples and sent it to a sink on another machine. That machine had a sink that read the data, performed some simple arithmetic operations, and discarded the data. After 30 seconds, the source machine notified the controller to log the current time ($t_1$) and terminated its process. The sink machine reported the time at which this failure was
detected \((t2)\). The controller then saved \(t2 - t1\) into a log file, and the mean detection times were computed from this log.

In Figure 5-5a, we see that setting \(n\) to 1 produces a high false positive rate, nearly 10\% for all values of \(s\) and \(k\). This most likely occurs because \(n = 1\) requires the sender and receiver to run in lockstep. However, the phase difference between the sender and receiver \((\Phi)\) is likely to drift over time, and the skew could grow large enough to generate a false positive. Hence, even though \(n = 1\) produces detection times of 20-30 ms, the high error rate makes it impractical. As \(n\) increases, we see a sharp drop in false positives. Most of the lines level out to 0\% false positive rates as \(n\) approaches 5.

In Figure 5-5b, we see the most promising results: detection times drop to around 50-60 ms for each line at 0\% false positive rates. The best performances come when \(k\) is fractional. When \(s = 20\) ms and \(k = 0.5\), detection times are 28.8, 38.6, and 48.8 ms for \(n = 3, 4, 5\), respectively. Each of these results had 0\% false positives.

When \(k\) is fractional, the receiver gives the sender multiple opportunities for a single keepalive to be sent, which helps mitigate the effect of skews in \(\Phi\) or other delays. If \(k \geq 1\), the receiver assumes that there will be a certain number of keepalives when it goes to check. Hence, there is a danger that some of those keepalives will be delayed.

As we have seen, reducing false positives can be accomplished by increasing the number of missed keepalives by raising \(n\). Again, we could also reduce the receive interval by decreasing \(k\), which has a similar effect as increasing \(n\), but at less of a cost to detection time.

\section{5.9 Detection under stress}

All the previous experiments were run when the software radio application was running as the main process on the system. Now we would like to see how fast failures can be detected on a loaded system, i.e., one that has another process consuming all
Figure 5-5: Graphs of accuracy vs. detection times varying sender interval (s), receive interval (ks), keepalives missed threshold (n). Each point shows the mean detection time and the overall false positive rate from running a test application 100 times.
the CPU time that it can. On uniprocessor machines, the Linux kernel will then split
the CPU time among the software radio application and other processes. The net
effect is that the keepalive scheduling will be delayed while the other process runs.

Figure 5-6 shows the results from failure detection times on a loaded system. We
see that there is a high rate of false positives (close to 50%) for \( n = 5 \). As we increase
\( n \) up to 25, we see that the false positive rate drops close to 0%.

These experiments suggest that it even if the software radio application must share
the CPU with another process, we can at least say that a failure can be detected within
300 ms. Detection time rises dramatically under stress because of the Linux scheduling
policy. Under the default scheduling policy, each process is given a maximum of time-
slice—about 200 ms as a baseline—for which it can run. The more time a process
has left, the higher chance it will be selected to run. When each runnable process
has consumed its time-slice, the kernel recomputes time-slices for each process. CPU-
intensive processes, liked the one used in these experiments, will consume their entire
time-slices immediately. Therefore, the keepalive sender and receivers may not get a
chance to run for at least 200 ms, which is why we see a high rate of false positives
under 200 ms. We can reduce false positives by raising \( n \), which will increase detection
time but reduce false positives.

5.10 Overhead

Figure 5-7 shows the CPU overhead for the keepalive system for one machine.
It measures the percentage of the time spent sending and receiving keepalives for
different time intervals over the entire process time. At the fastest interval, 10 ms,
the CPU overhead is highest at 0.18%. There is a sharp drop-off from 10 to 20 ms,
but the gains on reducing the intervals bottom out as the interval is increased to 80
ms.

The results show that the computational overhead of the keepalive system is very
low. This is what we would expect because even at the fastest interval, 10 ms, the
Graph of false positive percentages vs. detection times under CPU stress

Figure 5-6: False positives vs. detection times for loaded system

<table>
<thead>
<tr>
<th>s (ms)</th>
<th>r (ms)</th>
<th>n</th>
<th>expected (ms)</th>
<th>measured (ms)</th>
<th>difference (ms)</th>
<th>false positives (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td>5</td>
<td>50</td>
<td>94</td>
<td>44</td>
<td>46</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
<td>100</td>
<td>164</td>
<td>64</td>
<td>6</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>20</td>
<td>200</td>
<td>285</td>
<td>85</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>25</td>
<td>250</td>
<td>325</td>
<td>75</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>50</td>
<td>500</td>
<td>577</td>
<td>77</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>5</td>
<td>100</td>
<td>113</td>
<td>13</td>
<td>56</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>10</td>
<td>200</td>
<td>210</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>20</td>
<td>400</td>
<td>408</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>25</td>
<td>500</td>
<td>508</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>50</td>
<td>1000</td>
<td>1009</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>10</td>
<td>5</td>
<td>86.3</td>
<td>95</td>
<td>8.7</td>
<td>28</td>
</tr>
<tr>
<td>20</td>
<td>10</td>
<td>10</td>
<td>156.9</td>
<td>195</td>
<td>38.1</td>
<td>18</td>
</tr>
<tr>
<td>20</td>
<td>10</td>
<td>20</td>
<td>276.8</td>
<td>395</td>
<td>118.1</td>
<td>2</td>
</tr>
<tr>
<td>20</td>
<td>10</td>
<td>25</td>
<td>348.9</td>
<td>495</td>
<td>146.1</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>10</td>
<td>50</td>
<td>583.7</td>
<td>995</td>
<td>411.3</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.2: Experimental results for failure detection times under stress
keepalive system makes only a few system calls to write and read packets. Since none of these calls ever spin or block, the overhead is almost negligible. The application will be loaded by signal processing computations and hardly notice the effect of the detection mechanism.

![CPU Overhead of Keepalive Detection for One Sender and Receiver](image)

Figure 5-7: The CPU overhead of the detection system for one sender and receiver
Chapter 6

Related work

Related work has dealt with the checkpointing issues needed to make more sophisticated distributed software radio applications. Murakami has shown an implementation of a call processing system that reduces messaging logging and replaying needed by limiting checkpointing to certain events [23]. Other systems have focused on load balancing on highly available systems. Microsoft’s Tiger Project provides fault-tolerance by mirroring two copies of data onto different disks and scheduling a time for individual machines to deliver different parts of requested streams [4]. The SunSCALR framework, mentioned earlier, also uses keepalives to detect failures, but its main purpose is to provide services which can be hosted locally on a single machine, such as a Web and file servers [27].

Other work has attempted to provide fault-tolerance using CORBA. This work could be used to improve the reliability of the RDL application driver. One approach is to automatically deliver CORBA invocations to multiple machines to provide replication. The Horus and Electra projects provide a transparent mechanism to multicast CORBA invocations to process groups, which can be specified by the application writer [26].
Chapter 7

Conclusions and Future Work

This work shows the feasibility of detecting single failures in 50 ms on distributed software radio applications running on general-purpose, standard operating systems. To detect a failure quickly, the system schedules rapid keepalives with application-level system calls. The keepalive model presented shows that detection time is a function of the sender and receiver intervals and the number of keepalives that can be missed before a failure is signaled.

In theory, detection time is primarily limited by how fast the operating system can generate interrupts or schedule a thread for the keepalive mechanism, which is 10 ms in Linux. However, this investigation finds that a number of factors requires that the thresholds for the number of missed keepalives be increased to reduce the rates of false positives. First, the Linux kernel can schedule another process or thread to run for a short time while detection is occurring. In addition, the kernel can spend a few milliseconds in kernel space handling interrupts. Either case could cause the sender and receiver to delay the keepalive sender, causing the receiver to report a failure. Second, the lack of a real-time scheduling policy means that the keepalive sender and receiver could become out of sync, which could lead to a reported failure if the threshold for missed keepalives is low.

To further reduce detection times, lower-level mechanisms, like programming an interrupt generator, must be used. Lower-level mechanisms will accomplish two things. First, it will allow the system to schedule faster keepalive intervals. Sec-
ond, it will produce better timing guarantees. This mechanism should be less dependent on the operating system’s scheduling policy. This work also shows that sending keepalives at a rate of 10 ms produces little computational overhead, and we expect that this will hold true at even faster intervals.

7.1 Future Work

Future work involves improving many aspects of the failover system. Our efforts will focus not only on reducing detection times to under 10 ms, but also on enhancing the recovery mechanisms. Our goal is to construct a much more robust failover system that has the ability to optimize the number of backups that will be needed, preserve state during certain stages, and build support for application-specific recovery schemes.
Bibliography


68


