Detection Techniques for Fault Tolerance

Robert S. Hanmer
Lucent Technologies
2000 Lucent Lane 2H-207
Naperville, IL 60566-7033
hanmer@lucent.com
+1 630 979 4786

Abstract

Errors must be detected before they can be recovered or tolerated in highly available or highly reliable systems. These patterns discuss a time-honored method of detection, a Watchdog timer, and its larger cousin the System Monitor. This mechanism functions by having one part of the system watch for status or "heartbeat" messages to be delivered from the part being monitored. Other patterns help to make the monitoring and detection implementation efficient.

Introduction

The most desirable type of error handling in a computing system is when the error is detected automatically and corrected by the erroneous component. If this isn't possible then the next most favorable option is the "crash" failure mode. In a crash failure mode the failing element stops processing and does it without corrupting any of its peers. To prevent corruption of peers, the failing element stops without informing the world that it is stopping, thus it fails silently. [Saridakis]

Because a failing element might fail without notifying anyone else, some means of detecting that it has stopped after detecting an internal failure is needed. A WATCHDOG is a time-honored way of watching parts of the system to detect that they have failed silently.

Watchdogs have been used for many years in microcontroller-based systems. In these systems, there is frequently a processing element that acts strictly as the Watchdog element. This Watchdog will invoke a system reset lead to restart the microcontroller if it detects an anomaly.

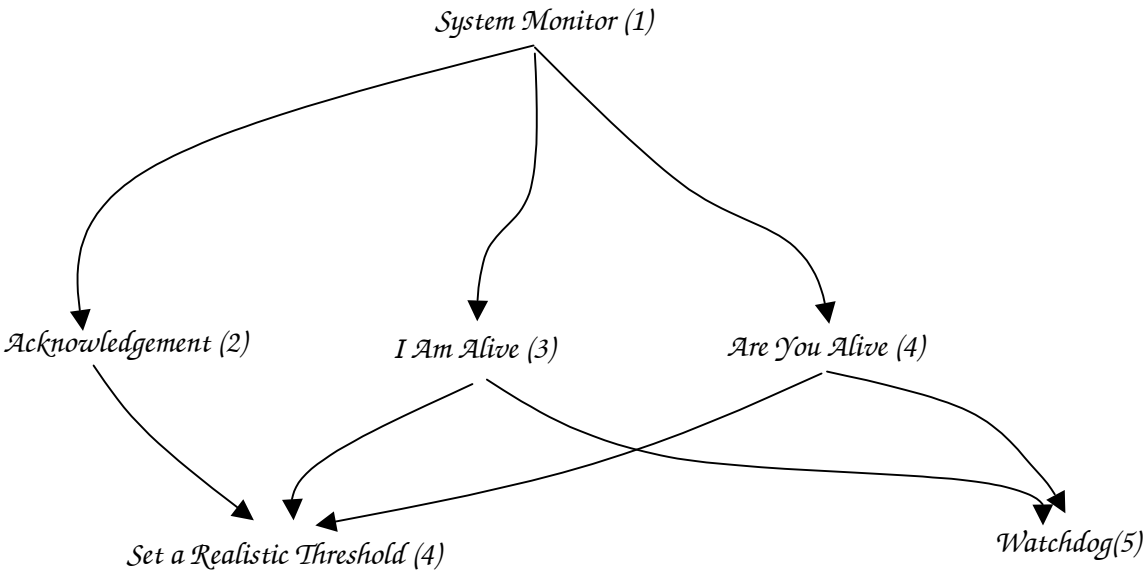
Watchdogs are also useful in systems built with components larger than microcontrollers. But in these larger systems the Watchdog elements are usually enhanced by software components that oversee the monitoring. The Watchdog becomes one of the tools at the disposal of the monitor.

The patterns here reflect this larger component case of a system. Patterns discuss both Watchdog and System Monitoring patterns. They also address the architectural and design choices that a system must make to monitor and detect crash failures.

These patterns are targeted at systems with extreme requirements of either high availability or high data integrity. In either situation if a part of the system fails you want to know about it quickly so that it can be restored to service to enhance availability, or isolated and kept from service to prevent data corruption.

The patterns presented in this paper duplicate several patterns from [Saridakis], placing them in the Alexandrian form. They are needed in this form to take their place in a larger collection of patterns.

The term "task" will be used here to denote an element of computation that could be the single program in a microcontroller or a process or a thread or any other way of grouping execution time into specific application functionality.



1. SYSTEM MONITOR

… The system that we are designing is expected to be available continuously. In the event that part of it stops working you need to find that out quickly so that the situation can be remedied. The system might exhibit crash failures, where it silently stops working.

The characteristics of the communications systems, such as paths and performance characteristics, are known. We can add additional messages without harming system behaviour by saturating the system.

Consider two domains of interest: space and time. Other parts of the system deal with the space domain, such as memory audits, database integrity checks and so on. Something is needed to make sure that the active parts of a system continue to operate correctly over time.

❖ ❖ ❖

You're designing a computing system and the architecture contains a critical component that cannot be allowed to deviate from correct operation. How do you keep track that it is alive and functioning properly?

Silently stopping working is desirable to the situation where the failing component is too aggressive about telling the world that it has a problem. Blindly reporting the problem to everyone uses up networking bandwidth and all the processing power, even though not all the parts of the system need to know.

An alternative is that the failing component just keeps "looking" like it is working. Performing actions but not performing them correctly. This is a significant problem because everyone might be fooled into thinking that the failing component is working correctly.

In the end the silent cessation of operations in a crash failure mode is the best for ensuring the system's continued operation. [LeeAnderson] But when a part of the system just stops, silently, the other parts might not realize, and might not take the steps that they need to to keep the application functionality working. Some way of finding out if a part of the system is still working is needed.

You could rely on acknowledgement messages. These messages are sent from one component to another in response to something happens. See the ACKNOWLEDGEMENT (2) pattern. They work well when you can't afford any messaging bandwidth or processing time to proactively monitor. A problem is that Acknowledgements require that there be an ordinary exchange of information that can stimulate the acknowledgements.

Another solution is to create a process that will actively monitor the task in question. It could be in the same processor or in a different processor. Choosing a location for this monitoring task is highly sensitive to the context. If the system is a simple one that must execute on a single processor and hardware support is unavailable, it must be located as a separate task on the same processor. If the system is a custom development with the option of adding a separate monitor hardware component, and the system's reliability requirements are sufficiently stringent then separate hardware is the way to go.

An important part of the monitoring task's purpose is to watch the tasks in question and report a problem if they stop working. It could monitor one important task, or it could monitor several tasks.

Therefore,

Create a task to monitor system behaviour, or the behaviour of specific other tasks in the time continuum, i.e. make sure that they are operating correctly. It should monitor the other parts of interest and report that it has stopped or take corrective action.

❖ ❖ ❖

Once you've decided to include a SYSTEM MONITOR you must decide how long it should watch before sounding an alarm. SET A REALISTIC THRESHOLD (5) discusses appropriate threshold times. RIDING OVER TRANSIENTS [Adams+] and LEAKY BUCKET COUNTERS [Adams+] discuss ways of making sure that the anomalous behaviour that you are seeing is an actual problem rather than a transient.

Refer to patterns about recovery and masking for suggestions of actions that can be taken if the SYSTEM MONITOR detects an error.

There are three ways that a SYSTEM MONITOR can operate. If additional resources (processing and messaging both) can be spared for the monitoring tasks the SYSTEM MONITOR can ask "ARE YOU ALIVE" (4) or it can monitor "I AM ALIVE" (3) messages from the monitored tasks. If resources cannot be spared, the perhaps specialized hardware can be added to the task from afar, just keeping track that it is behaving correctly. This is WATCHDOG (6).

Now that you've created a way to find out if a watched task is still functioning, what should you do when it isn't? The correct choice of action depends on the context. In a situation such as a financial application where numeric correctness is required, a desirable option to consider is shutting down the system. For example, it would be better for an Automatic Teller Machine to shut down than to erroneously dispense money. In a highly available situation where correctness is not as important, then there are three other options that could be employed: switching to a standby process/task, restarting the faulty task or just noting the situation and leaving the remedy to a human. …

2. ACKNOWLEDGEMENT

… The monitored task is in regular communications with the SYSTEM MONITOR (1) or with a 3rd entity that the SYSTEM MONITOR (1) can reliably share information with.

Detection doesn't need to occur immediately. For example, if we only need to know if a crash failure has occurred when we try to use the crashed, monitored task.

Information that a task has failed is only needed when the task is in use. If the task is unneeded the failure may remain latent and some other mechanism will detect and correct it.

Normal operations consist of a request (a query or instruction) from one task to another. It may or may not have a reply (an answer or a completion report). The target of the request (the task that will perform the action and possibly reply) is the system to be monitored. This could be either a client-server system or a system that requires peer-to-peer communication.

❖ ❖ ❖

What is the cheapest way to detect a failure?

You could add extra overhead by adding in separate mechanisms to report a failure, such as ARE YOU ALIVE (4) or I AM ALIVE (3) messages. But they add complexity and hence the potential for faults.

You could add hardware to act as a WATCHDOG (6) but that adds in its own level of complexity.

An easy way is to add some information to the reply that is already sent. Some additional complexity will be required in both parties. In the requester logic will be required that looks for valid information in the acknowledgement. The monitored system

might add some special information if it is aware that it is being monitored. If the monitored system is not aware that it is being watched then there might be no extra complexity required.

A disadvantage of using the reply is that if there aren't requests there aren't replies that can report a status.

If the communication protocol doesn't require a reply then this mechanism doesn't work, as there is no routine communication.

Therefore,

Include an acknowledgement requirement on all requests. All requests should require a reply to acknowledge receipt and to indicate that the monitored system is alive and able to adhere to the protocol. If the acknowledgement reply is not received then threshold and report a failure.

❖ ❖ ❖

Refer to SYSTEM MONITOR (1) for a discussion of actions that might be taken if the ACKNOWLEDGEMENT is not received. Refer to SET A REALISTIC THRESHOLD (5) for help setting the threshold values. …

3. I AM ALIVE

Suggestions welcome.

… Crash failures are possible. And they must be detected as quickly as possible.

Communications paths have known and finite communications latency. We know in advance how long messages will take to go between the SYSTEM MONITOR (1) and the monitored task. We can add additional messages without harming system behaviour by saturating the system.

In many situations the monitored task is responding to requests. If there aren't requests the mechanisms such as ACKNOWLEDGEMENT (2) won't work.

In this pattern, the monitored system wants others to know that it is functioning normally.

❖ ❖ ❖

How do you make sure that the monitored task is not erroneously marked as failed just because it is idle?

In situations where the monitored system is responding to requests, its activity level is dependent upon the request level. If the request level is low than acknowledgements won't be sent frequently (ACKNOWLEDGEMENT (2)). Building the intelligence to vary the thresholds (SET A REALISTIC THRESHOLD (5)) based upon request level adds complexity. But without it we don't know if the system is idle because there's no work or idle because it has had a crash failure.

There is a time overhead associated with adding messages that must be factored into any design decisions. During idle periods this will be negligible. During busy periods the REALISTIC THRESHOLD (5) is really important to reduce the unnecessary overhead that is draining processing and messaging bandwidth from real work.

If activity is low then it might look dead when idle. (A well-known human computer interface saying is "A quiet system is a dead system".) The monitored task must do something to let the monitors know that it is working. This could be to change a flag value occasionally, send a message to the monitor reporting health, display an hourglass icon, etc.

Therefore,

The monitored system should send a report to the SYSTEM MONITOR (1) on regular intervals. If the monitoring system fails to receive these reports it should threshold and then report that the monitored thing has stopped.

❖ ❖ ❖

The system must do something that can be watched. If the monitor detects that the action was taken, but doesn't look quite right, the monitor must decide if it was a fault in the communications system, or if the monitored system is alive, but not well. …

4. ARE YOU ALIVE?

Suggestions welcome.

… Crash failures are possible. And they must be detected as quickly as possible.

Communications paths have known and finite communications latency. We know in advance how long messages will take to go between the SYSTEM MONITOR (1) and the monitored task. The volume of communication in the system is below the saturation point (meaning that we can add additional messages without harming system behaviour).

This pattern applies when the monitored task has little interest in keeping the outside world aware that it is still functioning. It also applies when the monitored task can reply to a message but does not have the ability to initiate a message.

❖ ❖ ❖

How do we find out that the monitored task is no longer processing?

If the monitored task doesn't reply to a request for status from the SYSTEM MONITOR (1) it is assumed to have failed and action should be taken as described in SYSTEM MONITOR (1).

Sometimes the task being monitored doesn't realize that it is being watched. This can occur when an existing application is being employed in a highly available environment. When an existing application gains its new responsibilities, sometimes it cannot be altered, so it can't assume more of the monitoring relationship (such as I AM ALIVE (3) would have it do).

The monitored task might have such an important place in the system that it doesn't care that others know it is alive; it assumes that it always will be. This kind of task frequently arises when people unversed in fault tolerance and availability principles design without considering failure conditions.

A "ping" like message can be used to stimulate an Acknowledgement (2). This is a very simple way to check if the monitored task is still processing. Too many of these messages – both in frequency between any pair, and related to the number of monitored tasks can saturate the messaging system however.

Therefore,

The SYSTEM MONITOR (1) should send periodic requests for status to the monitored task. If the monitored task doesn't reply within the time required (see SET A REALISTIC THRESHOLD (5)), action to recover it should be taken.

❖ ❖ ❖

If the replies to the requests come, but doesn't look quite right, the monitor must decide if it was a fault in the communications system, or if the monitored system is alive, but not well. …

5. SET A REALISTIC THRESHOLD

… You are implementing a SYSTEM MONITOR (1) to monitor critical functionality. It is based on a task performing normal activities over time.

❖ ❖ ❖

How much time should elapse before the SYSTEM MONITOR (1) takes action?

Two different times are of interest, detection latency and messaging latency. Both require some care in choosing their values. *Detection latency* refers to how long the SYSTEM MONITOR (1) should wait for no response and the appearance of death before taking action. *Messaging latency* refers to the time between expected queries as to whether the monitored task is alive. Detection latency values are typically expressed as an integer multiple of the Messaging latency. For example, the detection latency might be N missed messages and each of these N messages is sent "messaging latency" apart.

Poorly chosen values of either detection latency or messaging latency will result in poor system performance. If either the messaging latency or the detection latency is too short the SYSTEM MONITOR (1) is hypersensitive to minor hiccups. Too long and the effectiveness of the SYSTEM MONITOR (1) is diminished, because too many errors can sneak through before the error is detected.

There are several key relationships that must be considered: length of time of a restart, communications round trip time and the severity of potential problems that go undetected.

You have some sort of availability requirement that gives you the final answer to this problem. For critical tasks, without an active load sharing companion, the SYSTEM

MONITOR (1) cannot afford to wait so long that one failure + the detection latency time + the time to restart the task exceeds the availability requirement.

If the task being monitored has parallel instances that can share the load while it is not working, especially with automatic load distribution by some other entity, then the detection latency time can be longer.

The messaging latency in this pattern is almost always slightly greater than the communications round trip time. One reason is that having an excessive number of status messages in the system is considered a drain on resources, so generally we should wait for a message to return or time-out before sending the next message.

A minimal messaging latency is the worst-case communications round trip time + the processing time at the component being monitored + the processing time at the SYSTEM MONITOR (1). This requires the SYSTEM MONITOR (1) to be waiting for and processing only one status message at a time.

A minimal detection latency value is one messaging latency interval. Setting detection latency to be equal to one messaging latency interval makes a very unforgiving system. Any errors at all will trigger an error report. Unexpected communications messaging latencies as well as slow target processing can result in errors being reported when the situation does not truly indicate that the target has failed.

Typically however the detection latency is set to be a small number of messaging latency-times. For example, an error might be reported when the three status messages have been missed.

Therefore,

Set the messaging latency based upon the worst-case communications time combined with the time required to process one heartbeat message.

Set the detection latency based upon the criticality of the functionality. Make it a multiple of the messaging latency. A small multiple for extremely critical or unique tasks, longer for duplicated tasks.

❖ ❖ ❖

Refer to SYSTEM MONITOR (1) FOR a discussion of what steps you can take when the error is detected.

Techniques such as RIDING OVER TRANSIENTS [Adams+] and LEAKY BUCKET COUNTERS [Adams+] can help smooth responses and make even too short detection latency work well. …

6. WATCHDOG

… You need to monitor the system and you're going to implement a System Monitor (1). You are worried about adding complexity because that can reduce the reliability.

There are a number of different contexts in which this pattern will apply. They represent different dimensions in your architecture, and not all of them need to be met for this to be the correct pattern: You can't spare messaging cycles to add new messages that you'd have to add if you implemented ARE YOU ALIVE (4) or I AM ALIVE (3). You might have the ability to add special hardware to implement the monitoring functionality.

❖ ❖ ❖

How can you ensure a task is alive when you want to streamline the mechanism or you can't or don't want to spare messaging resources?

The first thought is to have the monitored function report its status. The patterns ARE YOU ALIVE (4) and I AM ALIVE (3) add extra messages, or at least extra information to existing messages to report that the monitored task is actually working. But in many circumstances the messaging bandwidth has already been undersized for the application and you just can't add in any new messages. And if something is faulty it may not accurately report its state correctly. It also adds complexity, and hence the chance for additional faults.

If you can't add messages, you could add checks on the validity of the monitored tasks operations. Perhaps verifying the results with another copy (N-Version Programming), or matching with another processor. But these techniques are very expensive, both in terms of processing resources and complexity. Extra complexity adds extra faults.

One way of watching the system without increasing the complexity is to watch things already happening. For example, watch to ensure that the communications is happening as expected. Are messages going both ways? Frequently enough?

Sometimes you can add new hardware to the design. A simple control that will watch the monitored tasks execution can be built. It might monitor the control leads of a microcontroller, or it might watch a word in memory that a task is known to monitor, or it could be a passive observer watching an exchange of messages.

Even if new hardware cannot be added the monitor can watch some the same kinds of things from another process on the same processor or even from another general-purpose processor (as contrasted with dedicated watching hardware).

Therefore,

Add in the capability for the monitor to observe the monitored tasks activities, much as a Watchdog tends the flock. The Watchdog should take some actions to get the monitored task back on track if it strays to far from expected and desired behaviour.

❖ ❖ ❖

If there's normally (and expectedly) a low level of activity from the monitored task then perhaps a more energetic technique such as ARE YOU ALIVE (4) or I AM ALIVE (3) is needed to meet detection requirements. If there's a low level of activity then the extra messaging that they require might be manageable.

Refer to SYSTEM MONITOR (1) for some ideas of the kinds of things that a WATCHDOG can do if it detects that the monitored task is not behaving properly.

Refer to Patterns for Time Triggered Embedded Systems [Pont] and [PontOng] for much more information about using Watchdogs in the microcontroller domain.

An example of watching existing indicators is found in OVERLOAD ELASTICS [Hanmer00].

Acknowledgements

All photographs used courtesy of the United States National Archives and Records Administration, *www.nara.gov*.

Michael Pont offered many valuable comments and suggestions as shepherd that significantly improved the organization of these patterns. Mark Bradac helped as a local reviewer/sounding board for the patterns. Lars Grunske also reviewed a draft.

References

[Douglass99] Douglass, B. P. Doing Hard Time. Reading, MA: Addison-Wesley. 1999.

[Douglass02] Douglass, B. P. Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems. Boston: Addison-Wesley. 2002.

[Hanmer00] *Real Time and Resource Overload*. Proceedings of PLoP 2000 conference.

[LeeAnderson] Lee, P. and T. Anderson Fault Tolerance Principles and Practice, 2d edition. New York: Springer-Verlag Wien. 1981.

[Pont] Pont, M. Patterns for Time-Triggered Embedded Systems: Building Reliable Applications with the 8051 Family of Microcontrollers. New York: ACM Press. 2001

[PontOng] Pont, M. and R. Ong. *Using Watchdog Timers to Improve the Reliability of Single-Processor Embedded systems: Seven New Patterns and a Case Study*. Proceedings of VikingPLoP 2002, pp 159-200

[Saridakis] Saridakis, T. *A System of Patterns for Fault Tolerance*, Proceedings of EuroPLoP 2002, pp 535-582.

[POSA2] Schmidt, D., M. Stal, H. Rohnert, and F. Buschmann. Pattern -Oriented Software Architecture Volume 2 -- Patterns for Concurrent and Networked Objects. West Sussex: John Wiley and Sons. 2000.