A Fault Tolerance Analysis of Safety-Critical Embedded Systems

Project Report for the Completion of the Master of Science in Electrical and Computer Engineering at Carnegie Mellon University

Jennifer Morris

Table of Contents

Acknowledgments

I would like to thank my faculty advisor, Dr. Philip Koopman, for his guidance in this and all other work I have completed at Carnegie Mellon University. My earnest thanks also extend to Dr. Daniel Kroening for his assistance in this research, to Dr. Priya Narasimhan for her counsel, to Beth Latronico and Charles Shelton for their advice, and to my family for their support.

This work is supported in part by the General Motors Collaborative Research Laboratory at Carnegie Mellon University, the Pennsylvania Infrastructure Technology Alliance, and by Bombardier Transportation.

This material is based upon work supported under a National Science Foundation Graduate Research Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

This research was sponsored by the Semiconductor Research Corporation (SRC) under contract no. 99-TJ-684, the National Science Foundation (NSF) under grant no. CCR-9803774, the Office of Naval Research (ONR), the Naval Research Laboratory (NRL) under contract no. N00014-01-1-0796, and by the Defense Advanced Research Projects Agency, and the Army Research Office (ARO) under contract no. DAAD19-01-1-0485.

Abstract

Safety-critical distributed embedded systems often have strict dependability requirements that are influenced by other properties such as cost and size constraints, or the system architecture. For example, an automobile braking system might be required to tolerate a single fault in any component, as well as maintain some level of system availability. One fault-tolerance strategy might be to distribute processes over multiple nodes, use redundant brake hardware, and isolate the braking system's communication network from other components in the automobile. If this same system, however, is prohibited by cost and weight constraints from using redundant hardware, a different strategy must be used.

When cost and efficiency concerns require some safety-critical distributed embedded systems to use a single network for both critical and non-critical messages, particular care must be taken to ensure that faults in the non-critical components do not corrupt critical nodes. One threat to system dependability in this type of system is a software defect masquerade fault, where a software defect causes one node or process to send a message as having come from another node or process. The first chapter of this report outlines what software defect masquerade faults are and why they are often ignored in current embedded systems. It also includes preliminary research into methods to prevent them in embedded system design. The second chapter describes six successively more expensive levels of protection against masquerade faults. In addition, it presents a technique for preventing arbitrary, non-malicious masquerade faults that uses lightweight digital signatures based on CRCs. This technique is a lower-cost alternative to expensive, full-strength cryptographic security.

Some safety-critical distributed embedded systems may need to use centralized components to achieve certain dependability properties. The difficulty in combining centralized and distributed architectures is achieving the potential benefits of centralization without giving up properties that motivated the use of a distributed approach in the first place. The third chapter of this report examines the impact on fault tolerance of adding selected centralized components to distributed embedded systems, and possible approaches to choosing an appropriate configuration. It analyzes the proposed use of a star topology with centralized bus guardians in the Time-Triggered Architecture by modeling systems with different levels of centralized control in their star couplers, and comparing fault tolerance properties in the presence of star-coupler faults.

Chapter 1. Software Defect Masquerade Faults in Distributed Embedded Systems

J. Morris and P. Koopman. "Software Defect Masquerade Faults in Distributed Embedded Systems", In IEEE Proceedings of the International Conference on Dependable Systems and Networks Fast Abs, San Francisco, June 2003, pp. B50-B51.

Distributed embedded systems often consist of multiple nodes that communicate over a shared network. For such systems, dependable message delivery among nodes is crucial to overall system dependability. One threat to this dependable message delivery is a *software defect masquerade fault*, where a software defect causes one node or process to send a message as having come from another node or process. Unfortunately, many embedded system designs do not address this particular failure mode. This chapter outlines what software defect masquerade faults are and why they are often ignored in current embedded systems. It also presents preliminary research into methods to prevent them in embedded system design.

1.1. What is Software Defect Masquerading?

A masquerade is "in authentication, the pretense by an entity to be a different entity" [1.1]. In distributed embedded systems, masquerading usually occurs when one node or process sends a message across the embedded network that only another node or process is authorized to send. Although masquerading in distributed embedded systems can be intentional (i.e., a malicious attack from an intruder), it may also result from a defect in the system itself. This usually occurs at the application level because resource constraints in many distributed embedded systems require the use of either a very simple operating system or none at all. In these systems the messages are constructed and sent directly by the application software.

In event-triggered networks that use a header field to identify the message sender, a node or process may masquerade by sending a message with a header field of another node or process. Time-triggered networks do not necessarily require headers if each node or process is allocated a particular transmission slot. In these networks, masquerading may occur if one node or process sends a message in the transmission slot of another node or process.

1.2. Why is it Overlooked?

Traditionally, masquerading has been viewed as a malicious attack, rather than a fault tolerance problem. [1.2] looked at malicious masquerade attacks in open distributed agent-based systems and suggest strong cryptography as a method for preventing such attacks. Embedded systems, however, are typically closed (i.e., their networks are not accessible to outside intruders) and usually do not include security methods for preventing malicious attacks. In fact, most fault tolerance techniques used in embedded systems not only fail to prevent masquerading, but also assume fault models in which masquerade faults do not occur. For example, the Byzantine fault model discussion in [1.3] assumes that the identity of each general is correct. If software defects within the system itself cause masquerading that invalidates these assumptions, then the fault tolerance techniques may or may not work.

Even if software defect masquerade faults are included in an embedded system design's fault model, the current methods for combating malicious masquerade attacks are often not practical to implement in most embedded systems. Authentication techniques such as strong digital signatures, which are designed to withstand malicious attacks and cryptanalysis, provide more security than is required to protect against inadvertent software defect masquerade faults, but require high processing and network overhead. Cost-focused embedded system designs often do not have enough resources for such strong cryptographic protection. For example, the eight bit microcontrollers that are prevalent in embedded systems are simply not designed to accommodate the complicated computational algorithms required to produce strong digital signatures.

1.3. How Can it be Prevented?

Although the assumption of a closed network is reasonable for most distributed embedded systems, a dependable embedded system should still provide protection against masquerading caused by software defects. The challenge is to find a method that will not add an unreasonable burden to product cost.

Many embedded systems developers attempt to mitigate the effects of software defects by dividing the embedded software into critical and non-critical components. Critical software is developed with an expensive, rigorous design process and is presumed to function correctly, whereas non-critical software is developed with a less expensive, less rigorous design process and is presumed to function incorrectly (for the purposes of safety cases). This approach provides adequate protection against software defect masquerade faults in the critical components, provided that the critical and non-critical software are not transmitting messages over the same network. On a shared network, however, non-critical software masquerading of critical nodes and processes might still occur.

One way to prevent software defect masquerade faults on time-triggered networks is to use a bus guardian. Bus guardians are modules connected to each node on the network that prevent the node from sending messages outside of its designated transmission slot. Even if a node has a software defect that causes it to send messages in another node's time slot, the bus guardian will prevent the message from being sent at the wrong time. Masquerading may still occur if there is a defect in the bus guardian (incorrect synchronization, guarding the wrong time slot, etc.); however this method provides more protection than relying solely on the node itself to transmit correctly.

Another possible solution is to create a lightweight digital signature that uses relatively few system processing and bandwidth resources, yet is powerful enough to prevent software defect masquerade faults. Security research traditionally focuses on creating strong cryptographic protection. However, embedded systems, which have a more relaxed fault model but more rigid resource constraints, could benefit from lightweight cryptographic techniques for non-malicious fault scenarios.

One preliminary technique we have developed is to use a modified cyclic redundancy check (CRC) as a lightweight digital signature. Many distributed embedded system designs utilize an application-level CRC to verify message integrity. Although CRC's are useful at providing protection against random bit errors due to hardware malfunction or environmental interference, a CRC in its usual form has inadequate software defect masquerade coverage. In most implementations, the same CRC polynomial is used for multiple nodes on the network. For example, an embedded network for automotive or rail applications may use one or more CRC's for each message, but every message will use the same CRC polynomial(s) to generate the frame check sequence (FCS). If every node or process uses the same polynomial to generate the CRC, then the sender of a particular message cannot be distinguished by the CRC alone.

The CRC is converted into a lightweight digital signature by using a different CRC polynomial or a different seed value for each node to generate the CRC. The seed value is the initial value of the CRC register used in the calculation (all ones or all zeros in most CRC implementations). By setting this number to a distinct value for each node, the resulting CRC may be used as a lightweight digital signature, ensuring that software from any particular node cannot forge CRC values to masquerade as a different node. Further details on this method can be found in Chapter 2.

1.4. References

[1.1] Longley, D., Shain, M., and Caelli, W. *Information Security: Dictionary of Concepts, Standards and Terms*, Macmillan Publishers Ltd., Basingstoke, 1992.

[1.2] Minsky, Y., van Renesse, R., Schneider, F.B., and Stoller. S.D. "Cryptographic Support for Fault-Tolerant Distributed Computing". *Proc. 7th. ACM SIGOPS European Workshop*, 1996, pp. 109-114.

[1.3] Lamport, L., Shostak, R., and Pease, M. "The Byzantine General's Problem", *ACM Transactions of Programming Languages and Systems,4,3*, 1982, pp. 382-401.

[1.4] Morris, J. and Koopman, P. "Critical Message Integrity Over a Shared Network", *International Conference on Fieldbus Systems and their Applications*, July 2003.

Chapter 2. Critical Message Integrity Over a Shared Network

J. Morris and P. Koopman, "Critical Message Integrity Over a Shared Network", In Proceedings of the 5th IFAC International Conference on Fieldbus Systems and their Applications, July 2003, pp. 145-151.

Distributed embedded systems often contain a mixture of critical and non-critical software processes that need to communicate with each other. Critical software is "software whose failure could have an impact on safety, or could cause large financial or social loss" [2.1]. Because of the high cost of failure, techniques such as those for software quality assurance described in IEEE Std 730-1998 [2.2] are used to assure that such software is sufficiently defect free that it can be relied upon to be safe. However, such a process is expensive, and generally is not applied to non-critical system components.

In a typical safety-critical transportation system, such as in the train or automotive industries, it is generally assumed that critical components will work correctly, but that non-critical components are likely to have defects. Defects in non-critical components could, if not isolated, compromise the ability of critical components to function. Thus, the simplest way to assure system safety is to isolate critical and non-critical components to prevent defects in non-critical components from undermining system safety. Such separation typically involves using separate processors, separate memory, and separate networks.

Separation of critical and non-critical networked messages (i.e., through the use of separate buses) can double the required network costs, both in cabling and in network interface hardware. There is strong financial incentive to share a single network between critical and non-critical message traffic. But, when such sharing occurs, it is crucial that there be assurance that non-critical network traffic cannot disrupt critical network traffic. This chapter assumes that non-critical network hardware is designed to be fail-safe. For example, in railroad signaling lack of message delivery leads to a safety shutdown, so safe operation is viable with off-the-shelf networking hardware. But other challenges remain to implementing such systems.

A significant challenge in mixed critical and non-critical networks is ensuring that a non-critical process is unable to *masquerade* as a critical message sender by sending a fraudulent critical message. Masquerading is considered malicious if an internal or external attacker intentionally represents itself as a different entity within the system; however, masquerading may also occur due to non-malicious transient faults and design errors that inadvertently cause one node or process to send a message that is incorrectly attributed to another node or process.

A *software defect masquerade fault* occurs when a software defect causes one node or process to masquerade as another [2.4]. One example is a software defect that causes one process to send a message with the header identification field of a different process. Another example is a software defect that causes one node to send a message in another node's time slot on a TDMA network. A software defect masquerade fault is not caused by transient anomalies such as random bit flips, but rather is the result of design defects (e.g., the software sends the message with the incorrect header x instead of the correct header y). Fault tolerance methods designed to catch random bit flips may not sufficiently detect software defect masquerade faults.

This chapter describes six successively more expensive levels of protection that can be used to guard against masquerade faults. Rather than limiting the analysis to malicious faults, the gradations presented recognize that many embedded systems have reasonable physical security. Therefore it is useful to have design options available that present tradeoff points between the strength of assurance against masquerading faults and the cost of providing that assurance.

2.1. Fault Model

Network fault detection techniques vary from system to system. Some rely solely on a network-provided message Cyclic Redundancy Code (CRC) or other message digest such as a checksum for error detection. Some critical applications add an additional application-generated CRC to enhance error detection. These techniques can be effective at detecting random bit errors within messages, but they might not detect erroneous messages caused by software defects that result in masquerading. This is especially true in broadcast-oriented fieldbuses in which applications have control over message ID fields and can send incorrect IDs due to component defects. In the worst case, these erroneous messages could lead to masquerading of critical messages by non-critical processes or by failed critical hardware nodes.

In order to determine the safeguards necessary to ensure correct behavior of critical components over a shared network, we must first understand the types of failures that can occur, as well as their causes. The strongest fault model, which includes malicious and intentional faults, assumes that an intruder is intentionally falsifying message traffic and has significant analytic abilities available to apply to the attack. Such malicious faults can only be detected by application of rigorous cryptographic techniques. Because a malicious attack is the most severe class of fault, such measures would also provide a high degree of fault tolerance for software defect masquerade faults. But full-strength cryptographic techniques are cost-prohibitive in many embedded systems. In systems for which malicious attacks are not a primary concern, lighter-weight techniques that protect against accidental (non-malicious) faults due to environmental interference, hardware defects, and software defects are highly desirable.

The simplest accidental failures come from random bit errors during transmission. In general, these errors are easy to detect using standard error detecting codes such as CRCs that already exist on most networks.

Errors due to software defects or hardware faults in transmitters are more difficult to guard against. They can result in undetectable failures in message content unless application-level error detection techniques are used because faults occur before the message is presented to the network for computation of a CRC. A particularly dangerous type of error that could occur is an incorrect message identifier or application-level message source field, which would result in a masquerading fault.

In an embedded system with both critical and non-critical processes, masquerading faults can occur in three different scenarios: (1) a critical process might be sending a critical message to another critical process; (2) a critical process might be sending a critical message to a non-critical process; and (3) a non-critical process might be sending a non-critical message to a critical process. Because critical processes are trusted to work correctly, critical messages are assumed to have correct information, and messages from non-critical processes are suspect. Safety problems due to masquerading can thus occur if a non-critical process sends one of the first two types of messages (i.e., if a non-critical process sends a message falsified to appear to be a critical message). An additional situation that is sometimes of concern is if a critical node suffers a hardware defect that causes it to masquerade as a different critical node, resulting in a failure of fault containment strategies (designs typically assume not only that faults are detected in critical nodes, but also that they are attributed to the correct critical node).

2.2. Protection Levels

Once the fault model has been defined, an appropriate level of masquerading fault detection can be implemented based on the needs and constraints of the application. As with any engineering design, this requires tradeoffs in cost, complexity and benefits.

2.2.1. Level 0 - Network protection only

The first, baseline, level of protection is to rely solely on network error checking. Most networks provide some mechanism to detect network errors. Ethernet and the Controller Area Network (CAN), for example, both use CRCs.

The problem with relying on the network-level data integrity checks is that they only check for errors that occur at the network link level. Errors due to software defects or some hardware defects in the network interface are not detected. In addition, these detection techniques may not be very effective at detecting errors caused by routers and other networking equipment on multi-hop networks. For example, Stone and Partridge [2.7] found high failure rates for the TCP checksum, even when combined with the Ethernet CRC.

Though relatively effective at preventing random bit errors, Level 0 remains vulnerable to defects in networking hardware that cause undetected message errors, software defects that result in masquerading by critical and non-critical processes, and malicious attacks. In terms of bandwidth and processing resources, this level requires no additional cost because it is already built into most networks.

2.2.2. Level 1- Application CRC

The next step in assuring message integrity is to apply an application-level CRC to the data and to include it in the message body that is transmitted on the network. Stone and Partridge [2.7] strongly recommend using an application-level CRC to help detect transmission errors missed by the network checks due to defects in routers and other networking equipment.

It might be the case that some or all of the processes within a system use the same application-level CRC. If non-critical processes use the same application-level CRC as critical processes, then the application CRC provides no protection against masquerading by non-critical processes.

Level 1 provides additional protection against defects in networking hardware that cause undetected message errors. However, it does not protect against critical message sources that falsify message source information due to faults, defects that result in masquerading by critical and non-critical processes, and malicious attacks.

With respect to resource costs, Level 1 requires some additional bandwidth and processing resources, but not much. For example, on a CAN network a 16-bit application CRC requires two of the eight available data bytes and an additional few instructions per data bit of CRC. In critical systems application-level CRCs are not uncommon.

2.2.3. Level 2 - Application CRC with secret polynomial/seed (symmetric)

The application-level CRC in Level 1 may be converted from a simple data integrity check into a lightweight digital signature by using different CRC polynomials for different classes of messages. In this scheme, there are three separate CRC polynomials used: one for critical messages sent between critical processes, one for non-critical messages sent by the critical processes to non-critical processes, and one for messages sent by the non-critical processes. It is important to select "good" polynomials with an appropriate Hamming Distance for the lengths of messages being sent, of course [2.6].

The Level 2 approach is a "lightweight," symmetric digital signature in which the secret key is the CRC polynomial. It is symmetric because both the sender and receiver of a message need to know the same key, and must use it to sign messages (by adding an application-level CRC using a specific polynomial), and verify signatures (by computing the application-level CRC using an appropriate polynomial based on the purported message source and comparing it to the frame check sequence (FCS) field of the message actually sent).

A straightforward implementation involves using a different secret polynomial for each class of message: CRC_1 for critical to critical messages; CRC_2 for critical to non-critical messages; and CRC_3 for non-critical message senders. (Note that the case where non-critical processes omit an application-level CRC is equivalent to using a null CRC for situation CRC_3).

Use of three CRCs is required because this is a symmetric system. Thus, it is possible that any process possessing a CRC polynomial might send a message using that polynomial due to a software defect. If CRC_1 is only known to critical processes, that means it is impossible (or at least probabilistically unlikely) that a non-critical process can falsify a message that will be accepted by a critical process as having come from another critical process. In other words, CRC_1 is a secret symmetric key, and only key-holders can generate signed messages. CRC_2 is used to provide assurance that critical messages are being sent either from critical processes or non-critical processes (with software defects) that are receivers of critical messages. CRC_3 is simply an application-level CRC for non-critical messages. It might be the case that there is no point in distinguishing CRC_2 from CRC_3 depending on failure mode design assumptions, because in either case at least one non-critical process would have access to the secret key CRC_2 for generating critical-process-originated messages.

With this scheme there is still a critical assumption being made about non-critical code. However, it is a much narrower assumption than with the Level 1 approach, and is probably justifiable for many situations. The assumption is that CRC_1 has been selected from a pool of candidate CRCs at random, and is unlikely to be used by non-critical processes on a statistical basis. (One assumes that "well known" published CRCs are omitted from the potential selection pool, of course.) For 24-bit or 32-bit CRCs this assumption is probably a good one, but there is still a finite number of "good" CRC polynomials that are significantly fewer than all possible 24-bit or 32-bit integers.

A solution that is even better for these purposes is to use a "secret seed" for a given polynomial. Conventional CRC calculations use a standardized starting value in the CRC accumulator, typically either 0 or -1. A secret seed approach uses some different starting, or "seed" value for computation of the application-level CRC that varies with the class of message. So instead of CRC_1, CRC_2 and CRC_3 for the previous discussion, the technique would involve using the same CRC with $Seed_1$, $Seed_2$, or $Seed_3$, with each seed being a different secret number. Thus, the seed value becomes the secret key for a digital signature.

Thus, the FCS of a message with a level of criticality *i* would be computed as follows. If CRC(M,S) takes a message M with an initial CRC seed value S to compute a FCS, then:

$$FCS_i = CRC(M,S_i) \tag{2.1}$$

Critical to critical process messages would be authenticated by having critical processes use S_1 to compute and compare the FCS field. Since no non-critical process would have knowledge of S_1, it would be, for practical purposes, impossible for non-critical processes to forge a correct FCS value corresponding to a critical message. There would still be a chance of an accidental "collision" between the FCS values for two CRCs, but this is true of cryptographically secure digital signatures as well, and can be managed by increasing the size of the FCS as required.

Combining a secret polynomial with a secret seed is possible as well, of course, but does not provide a fundamentally different capability. It is important to note that CRC-based digital signatures are readily attacked by cryptanalytic methods and are ***not secure against malicious attacks.*** However, in a cost-constrained system it might well be reasonable to assume that non-critical components will lack cryptanalytic attack capabilities, and that software defects will not result in the equivalent of cryptanalytic attacks on secret CRC polynomials or secret seeds.

Symmetric-key CRC lightweight digital signatures of Level 2 provide the same benefits as application-level CRCs of Level 1. In addition, they provide protection against non-malicious masquerading by non-critical processes that results in acceptance of fraudulent critical messages. However, Level 2 does not protect against non-malicious masquerading of critical message sources by other critical message sources due to faults, and is inadequate protection against malicious attacks. The benefit of Level 2 is that it requires no additional processing or bandwidth to upgrade from Level 1.

2.2.4. Level 3 -Application CRC with secret polynomial/secret seed (asymmetric)

Symmetric CRC-based signatures ensure that non-critical processes cannot send critical messages to critical processes by accident. However, a software defect could still cause a non-critical process to masquerade as a critical process sending a non-critical message. (This is true because all noncritical processes possess the symmetric key information for receiving such messages). Additionally, Level 2 assumes that all critical processes are defect-free, providing no protection against masquerading by a critical process in the event of a hardware failure or software defect.

A further level of protection can be gained by using asymmetric, lightweight authentication. In this approach every process has a secret sending key and a public receiving key. The public receiving key is known by all processes, but only the sending process knows the secret sending key. In such a scheme every process retains the public receiving keys of all processes from which it receives messages (in general, this means it has the public keys of all the processes). But because each process keeps its transmission key secret, it is impossible for one process to masquerade as another.

Because embedded systems tend to use broadcast messages heavily, an implementation of full public-key encryption is impractical, so the method proposed here is tailored to a broadcast environment. Additionally, CRC-based authentication is used which is of course ***not secure against a cryptanalytic attack***.

One way to implement a private/public signature scheme is the following, using secret polynomials. This method may also be used in addition to the use of distinct CRCs or seeds for FCS computation as outlined in Level 2. If desired for cost and simplicity reasons, all non-critical to non-critical messaging can use a single standard polynomial, and only critical message sources need use the private/public key approach.

Each critical process has two CRC polynomials: CRC_1 and CRC_2. CRC_1 is a publicly known polynomial, whereas CRC_2 is a secret private polynomial. Every CRC_1 in the system is distinct per process. Every CRC_2 is the inverse of the corresponding CRC_1. Thus, the secrecy of CRC_2 depends on there being no code to compute an inverse polynomial in the system. Because computing inverse polynomials is performed using a bit-reverse operation (with adjustments to account for an implicit 1 bit within the polynomial in most representations), the validity of the assumption of secrecy is one that must be made in the context of a particular system design. However, computing inverse polynomials off-line and putting them in as constants within the system code avoids the presence of inverse polynomial code, and might well be a reasonable approach for systems that cannot afford the cost of full-strength cryptography. (The creation of stronger, but efficient, methods for asymmetric signatures is an open area for future research.)

A sending process S appends a signature X to a critical message M and its FCS field (X is not included in the FCS computation), where "|" denotes concatenation:

$$M \mid FCS \mid X \tag{2.2}$$

where:

$$X = CRC_2(FCS) \tag{2.3}$$

Receiving processes then verify the authenticity of the transmission by ensuring that:

$$FCS = CRC_1(X) \tag{2.4}$$

What this is doing is "rolling back" the FCS using an inverse CRC, CRC_2, to compute a signature that, when rolled forward through CRC_1, will yield the FCS. Because only the sending process knows the inverse CRC for its public CRC, no other process can forge messages.

This method protects against software and hardware defects that cause a process to send a message that should not be sent (e.g., forged source field or incorrect message identifier/type information).

This method is vulnerable to the following: malicious attacks using cryptanalysis (even without knowledge of the public CRC polynomial); software defects involving CRC code that computes CRCs "backwards" from the critical CRC computation (e.g., right-to-left CRC computations when the critical code is using a left-to-right shift-and-xor computation); and software defects in critical or non-critical software that compute the bit-reverse of a public polynomial and then use that as the basis for signing a message. While some of these defects could probably happen in real systems, the specificity of the defects required would seem to provide a higher degree of assurance than not using such a technique. As stated previously, this is an example of a simple lightweight signature technique; it is possible that future research will yield even better approaches to fill this niche in the design space.

If the system is originally designed at Level 1 or Level 2 with an application CRC, then there is an additional cost to compute and transmit the signature X. In a CAN network with a 16-bit signature X, this would be an additional handful of instructions per CRC bit and two bytes of the remaining six available data bytes.

2.2.5. Level 4: Symmetric cryptography

Levels 1 through 3 all use some form of CRC to detect masquerading errors due to defects in the non-critical software, and provide no credible protection against malicious faults. The next higher level of protection can be achieved through the use of cryptographically secure digital signatures. Although designed primarily for malicious attacks, such digital signatures can also prevent defective non-critical software components from forging critical messages. This can be accomplished via use of a Message Authentication Code (MAC), which is a keyed one-way hash function. A detailed description of MACs appears in Section 18.14 of Schneier [4.5].

Symmetric digital signatures must be sufficiently long to preclude successful malicious attacks via cryptanalysis or brute force guessing. Additionally, they take significant computational capability beyond the means of many embedded systems. However, a symmetric key approach is secure against malicious attacks unless the attacker compromises a node possessing a secret key. In the case that the attacker compromised a critical code, it would be possible to maliciously forge a message that apparently originated in any node in the system. Malicious attacks aside, a Level 4 approach has the same strengths and weaknesses as a Level 2 approach in that it is a similar general signature method, but using strong cryptography.

2.2.6. Level 5: Public-key digital signatures

Level 4 protection was analogous to Level 2 protection, but used cryptographically secure symmetric digital signatures. Level 5 is, in turn, generally similar to Level 3 CRC lightweight public key signatures, but uses cryptographically secure signature algorithms. Various public-key digital signature algorithms are described in Section 20 of Schneier [4.5].

A Level 5 approach provides protection from forgery of message sources to the limits of the cryptographic strength of the digital signature scheme used. Moreover, if a node is compromised by malicious attack, forgery of messages can only be accomplished with compromised node(s) as originators, because each node has its own distinct secret signature key. However, public-key methods are much slower than symmetric cryptography [2.3].

2.2.7. Tradeoffs

Each of these methods provides a certain level of fault protection; however, they each have a commensurate cost. The developers must decide what protection is required to attain safe operation, and adjust system design decisions on how much safety critical operation to delegate to computers based on budget available to provide protection against realistic masquerading threats. For example, a system with a fault model that includes software defect masquerade faults but excludes malicious attacks might chose Level 2 or Level 3.

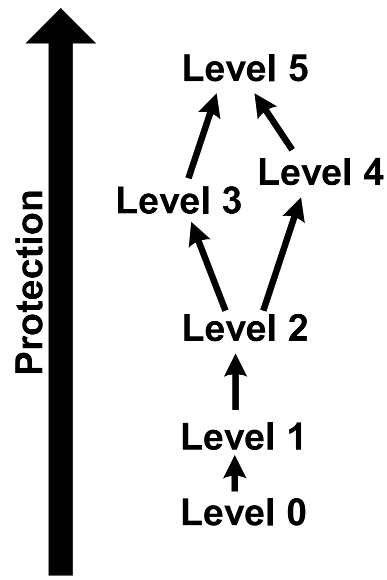

Figure 2.1: Masquerading fault protection levels

An additional burden that must be assumed when using any masquerading detection technique is that of cryptographic key management. Any technique discussed assumes that only a certain set of nodes have access to secret keys. This restricted access results in significant complications in configuration management, deployment, and maintenance, especially when insider attacks are considered a possibility. (As a trivial example, every time a disgruntled employee leaves a company, it is advisable to change all cryptographic keys that the employee might have had access to if attacks by that employee are a substantive threat.)

Figure 2.1 shows all of the levels, in order of effectiveness. In general, the stronger the protection, the more expensive the method. Levels 3 and 4 have a partial ordering, because the protection of Level 3 (asymmetric secret CRC) might be more useful than the protection afforded by Level 4 (symmetric secure digital signature), depending on whether malicious attacks are a part of anticipated threats. However, it is expected that CRC-based signatures will be substantially less expensive to implement than cryptographically secure digital signatures.

2.3. Conclusions

This chapter presents six levels of fault detection techniques that can be deployed against the possibility of masquerading faults on shared critical/non-critical fieldbuses. Level 0 (network-provided protection) provides no protection beyond what is included in the network protocol. For level 1 (published CRC), the application must be modified to apply the application-level CRC before sending messages on the network, and after messages have been received. Once an application-level CRC is present in the code, the polynomial or seed value used in the calculation can be changed to achieve Level 2 (symmetric secret polynomial/seed) protection. A novel Level 3 (asymmetric secret polynomial/seed) approach is proposed to provide very lightweight digital signatures with a public key flavor that are suitable for broadcast bus applications, but that further assume malicious faults are not a threat. Levels 4 and 5 complete the taxonomy and consist of using well known cryptographically secure approaches to guard against malicious masquerading faults.

Typical fieldbus systems today operate at Levels 0 and 1, and are not secure against masquerading faults. It might be attractive in some applications to upgrade to a Level 2 or Level 3 capability to improve resistance to non-malicious software defect masquerade faults without having to resort to the complexity and expense of cryptographically secure Level 4 or Level 5 approach.

2.4. References

[2.1] IEEE (1990). IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 610.12-1990.

[2.2] IEEE (1998). IEEE Standard for Software Quality Assurance Plans, IEEE Std 730-1998.

[2.3] A. J. Menezes, P.C. Van Oorschot, and Scott A. Vanstone, *Handbook of Applied Cryptography*. CRC Press LLC, Boca Raton, 1997.

[2.4] J. Morris. and P. Koopman. "Software Defect Masquerade Faults in Distributed Embedded Systems", In *IEEE Proceedings of the International Conference on Dependable Systems and Networks Fast Abs*, San Francisco, June 2003, pp. B50-B51.

[2.5] B. Schneier, *Applied Cryptography*. Second Edition, John Wiley & Sons, New York, 1996.

[2.6] D. P. Siewiorek and R. S. Swarz, *Reliable Computer Systems Design and Evaluation*. Second Edition. Digital Press, Bedford, MA, 1992.

[2.7] J. Stone and C. Partridge. "When the CRC and TCP Checksum Disagree", In *ACM SIGCOMM Computer Communication Review: Proc. of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, September 2000, pp. 309-319.

Chapter 3. Fault Tolerance Tradeoffs in Moving from Decentralized to Centralized Embedded Systems

J. Morris, D. Kroening, & P. Koopman "Fault Tolerance Tradeoffs in Moving from Decentralized to Centralized Embedded Systems", In IEEE Proceedings of the International Conference on Dependable Systems and Networks, Florence, Italy, to be published in June 2004.

Embedded systems designers consider many factors when choosing between distributed and centralized architectures. In some designs, centralized systems may be preferred for their manageability and consistency, whereas in others, distributed systems may be more advantageous for scalability and modularity [3.1].

In the context of dependability, distributed and centralized systems each may be better suited to deliver certain properties. A system with applications distributed across nodes that are physically separated may provide geographic fault isolation; one with a centralized controller may more easily maintain consistent timing and state information across all applications. A system with applications running on separate nodes must also ensure that those nodes have a consistent view of system state. Centralized controllers can become a single point of failure for all applications. The problem with choosing a design that is either strictly distributed or strictly centralized is that one approach might experience problems in system dependability that the other resolves, and vice versa.

In order to achieve the "best of all possible worlds," system architects may attempt to use a combination of centralized and decentralized components in one system. This approach has recently been seen in some designs based on the Time-Triggered Architecture (TTA). Researchers at the Vienna University of Technology recently proposed that the TTA with a star topology and central bus guardians, rather than a bus topology with guardians at each node, would eliminate the occurrence of some fault modes that are not tolerated by the Time Triggered Protocol (TTP/C) [3.2]. In their design, the central bus guardian could have the authority to: a) stop all nodes from transmitting outside of their assigned time slot to eliminate babbling idiot and masquerading faults, b) make adjustments to signal strength and frame timing in order to eliminate slightly- off-specification (SOS) faults [3.3], and c) perform semantic analysis of all frames transmitted on the network to prevent transmission of frames containing incorrect controller state (C-State) information [3.2].

Augmenting the authority of some system components, such as the bus guardians in this example, may help the system achieve certain dependability properties; however, faults in those more authoritative components may also have a greater influence on system dependability. For example, suppose a bus guardian suffers a fault that causes it to block transmission of all frames. In systems with decentralized bus guardians (*e.g.*, each node has a separate bus guardian), a fault of this nature in one bus guardian would only block frames from one node. The same fault in a central bus guardian would stop all nodes from sending frames on the channel. This particular fault mode is addressed in [3.2] by the use of redundant channels with separate central bus guardians. However, this example illustrates that changes in the behavior of one component may alter the way faults in that component affect system dependability.

Adding centralized components to a decentralized system may impact system properties beyond dependability as well. For example, suppose a central bus guardian is required to buffer some minimum number of bits of each frame in order to filter traffic on the network, and this number is proportional to the longest frame. If this central bus guardian is also prohibited from buffering all of the bits in the shortest possible frame, then the difference in length between the longest and shortest frames is limited. It might not be possible to send extremely long frames on the same network as extremely short frames.

Two questions that must be answered when centralization is added to a decentralized system are: “How much authority should the centralized components be given?” and, “what is the impact of this added authority on other parameters of the system?” To help analyze these questions we created a model in SMV (Symbolic Model Verifier) [3.4] of the TTA star topology with redundant star couplers. In the star couplers we modeled feature sets with varying amounts of centralized control. We also modeled the possible fault modes those features may exhibit. Our objective was to analyze the system-design tradeoffs associated with the centralized authority by modeling the failure modes that can arise when central guardians have the ability to buffer frames, and by showing how limiting the buffer size of a central guardian also limits the frame size and clock rates. Finally, we propose that critical systems with an active centralized hub for communication, such as a TTP/C network with a star topology, are constrained in clock speed by frame size, and vice versa.

The remainder of this chapter is organized as follows: Section 3.1 presents background information and previous work; Section 3.2 presents our objective and approach; Section 3.3 presents detailed information about our system model; Section 3.4 presents the results from modeling various star-coupler faults; Section 3.5 presents an analysis of those results; and Section 3.6 presents our conclusions.

3.1. Background and Related Work

The TTA is a distributed system architecture designed for safety-critical embedded systems such as automobiles and jet aircraft that consists of a cluster of nodes communicating over a shared TTP/C network. This section provides background information about TTP/C and the TTA, including the motivation for using central bus guardians in the star topology. We also describe our method for modeling these systems in SMV.

3.1.1. Overview of TTP/C and the TTA

The Time-Triggered Protocol (TTP/C) and the Time Triggered Architecture (TTA) were developed by researchers at the Technical University of Vienna and TTTech Computertechnik AG for use in safety-critical distributed embedded systems that require dependable frame transmission for hard real-time guarantees. To meet these requirements, TTP/C uses Time Division Multiple Access (TDMA), in which nodes send frames in specific time slots. These slots are statically assigned prior to system start-up in the Message Description List (MEDL) [3.5].

TTP/C provides a number of services, including distributed clock synchronization, group membership, and clique detection [3.5]. Clock synchronization is used by the protocol to enforce TDMA scheduling. Each node decides when to transmit on the network based on its view of the time; if different nodes have different views of the global time then the TDMA approach will fail. Group membership is provided as a safety service to allow host applications to monitor the operational states of other nodes on the network. The intention is that nodes that are not operating correctly can be removed from membership in the protocol until they exhibit correct behavior. Clique detection prevents multiple membership groups from forming.

These services are created by nodes generating and/or observing traffic on the network. Clock synchronization, for example, requires each node to observe frames sent by other nodes and calculate the difference between each frame's actual arrival time and the expected arrival time. This allows the observing node to adjust its own internal clock to bring it closer to the clocks of other nodes [3.5]. Group membership requires each node to verify that frames sent by other nodes are both valid and correct. A valid frame starts and ends during the time slot, exhibits no encoding rule violations, and is not interfered with by another transmission during the time slot [3.5]. Correct frames are valid frames that also have a controller state (C-state) and cyclic redundancy check (CRC) that match those of the receiving node [3.5]. The C-state information may be included in the frame explicitly or implicitly through its inclusion in the CRC calculation. If no activity is observed on the channel during the time slot, the frame is considered to be null (neither invalid nor incorrect).

The algorithms for deterministic message timing, group membership, and clique avoidance are claimed to be correct, given a set of particular constraints [3.5]. The fault hypothesis of TTP/C allows for an arbitrary failure in a single component of the system. The faulty component may be a single node, bus guardian, or channel. TTP/C as-

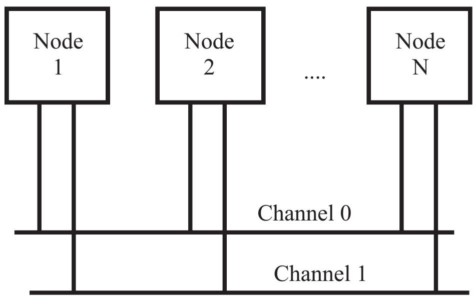

Figure 3.1: TTA Bus Topology, TTTech Computertechnik AG [3.5]

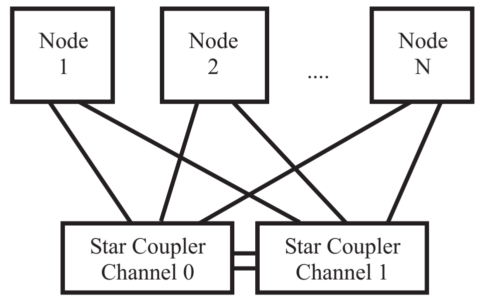

Figure 3.2: TTA Star Topology, TTTech Computertechnik AG [3.5]

sumes fail-silence in the time domain, meaning that a faulty controller will not be allowed to send frames outside of its time slot. TTP/C does not guarantee correctness of the application data contained in the frame.

Ensuring that these requirements are met is handled at the architectural level, as specified by the Time-Triggered Architecture (TTA). The TTA requires at least two independent buses (channels), between nodes. It is assumed that these channels themselves do not generate frames on the network; however, the channels may corrupt or drop frames. Bus guardians are also required in order to prevent faulty nodes from transmitting during the wrong time slot. These bus guardians must be completely independent of the nodes (separate internal clocking device, physical isolation, *etc.*) and may be allocated to nodes individually or centralized in a star topology. Fault tolerance for Byzantine faults [3.6] requires at least four real member nodes with fully independent bus guardians.

3.1.2. Central Bus Guardians

The Time-Triggered Architecture (TTA) may be implemented using either a bus (Figure 3.1) or a star (Figure 3.2) topology. In order to compare the fault-handling ability of the two topologies, Ademaj *et al.* [3.7] performed software implemented fault injection (SWIFI) and heavy-ion fault injection experiments on TTA systems with both bus and star topologies.

The experiments in the system with the bus architecture revealed several ways that faults in a single node could propagate to other, non-faulty nodes. First, slightly-off- specification (SOS) faults in the signal strength or timing of a frame can cause nodes on the network to have differing views of the validity and/or correctness of some frames [3.3]. A frame that is slightly outside of its time window or has a signal that is slightly below the acceptable range may be judged valid and correct by some nodes and invalid or incorrect by others. This is the result of slight differences in hardware tolerances between nodes. This disagreement between nodes on a frame's validity and/or correctness is an SOS fault.

Group membership is determined by checking the validity of sent frames; therefore a disagreement between nodes on the correctness of a frame could cause some nodes to expel that frame's sender from the group and others to keep the sender in the membership. If this occurs, the TTP/C clique detection algorithm detects the disagreement in group membership and causes the nodes in the minority clique to enter a freeze state. Nodes that have been frozen cannot regain membership and transmit on the network until they have been awakened by their hosts. This is the correct behavior of the protocol; however, frequent SOS failures could lead to frequent shutdowns of non-faulty nodes in the TTP/C cluster, which is unacceptable in most critical systems. Many of these critical systems, such as automobiles and jet aircraft, are fail-operational, and therefore require a high level of system availability.

Another fault mode in the bus architecture system is masquerading during cluster startup [3.7]. Prior to startup, nodes in the cluster have not yet established the global time, and cold-start frames that signal the start of a TDMA round do not arrive in a particular time slot; therefore, the sender of a cold-start frame cannot be verified by the arrival time of the frame [3.2]. If a faulty node sends a cold-start frame with an incorrect sender round slot, other nodes will attempt to integrate into the cluster at the incorrect time. If different cold-start frames arrive at different times on the two channels, nodes may try to integrate on either channel, the cliques will be discovered by the clique-detection algorithm, and nodes in the minority clique will go into the freeze state [3.2].

The fault injection experiments on the bus topology also revealed that frames with invalid C-states could cause problems for nodes integrating into a running system [3.7]. If a faulty node sends a frame with an invalid C-state, a non-faulty node that has already integrated determines that the frame is incorrect because the C-state of the frame does not match the internal C-state of the receiving node. Nodes that have not yet integrated do not know the correct C-state of the system, and therefore cannot recognize the frame as incorrect. During integration, they adopt the C-state of the first valid frame they receive and attempt to integrate using that C-state. If the C-state they adopt is incorrect, the integrating node will not be allowed to enter the active state, thereby preventing the perfectly operational node from integrating into the system.

In order to prevent SOS faults, masquerading during startup, and failures in integration arising from frames with invalid C-states, Ademaj *et al.* [3.7] implemented a star topology with the central bus guardians. In the new design, bus guardians located at the central hubs of the star topology are authorized to perform "active signal reshaping" of the transmitted frames [3.7]. The central bus guardians monitor frames on the network and boost signals that are SOS in the value domain and delay or block signals that are SOS in the time domain. The central bus guardians also perform semantic analysis of frame content to prevent faulty nodes from masquerading as other nodes during startup and/or transmitting frames with invalid C-states. Fault injection experiments of the new design showed that central bus guardians could prevent SOS faults from occurring. The experiments also showed that the central bus guardians could prevent faulty nodes from transmitting incorrect cold-start frames and frames with incorrect C-state [7].

3.1.3. Symbolic Model Checking

We use a symbolic model checker to analyze our formal models. Model checking is a systematic way to explore the whole state space of a formal model. Previous work containing formal models of TTP/C focuses on proving correctness of the protocol [3.8, 3.9,3.10]. This is done using an interactive theorem prover, which requires a considerable amount of manual work. In contrast to that, model checkers, such as SMV, are fully automatic once the model is written. The model presented in [3.10] contains more detail than the model we use. However, we are not aware of any formal models addressing the issue of centralized vs. distributed bus guardians.

3.2. Objective

The fault injection experiments of [3.7] showed that the central bus guardian could be effective at stopping faults on some nodes from propagating to other healthy nodes. However, these experiments did not analyze the behavior of the system in the presence of faults in the star couplers themselves. Once a component in a decentralized system has been given increased authority over other system components, the system must be reevaluated to ensure that unintended behaviors do not emerge when the centralized authority fails.

For example, TTP/C assumes that faults in either of the two channels are passive. That is, a channel will either corrupt or drop frames, but it will not generate frames. However, this may no longer be a valid assumption if the central bus guardian is given certain authorities. For example, if the central bus guardians have the ability to buffer entire frames, a fault in the central guardian could cause it to transmit frames outside of their intended time slots. If this occurs, the assumption of passive faults in the guardian no longer holds, because the bus guardian is essentially "generating" frames on the network.

In their analysis of failure mode assumption coverage in the TTA, Bauer *et al.* [3.11] assert that in order to maintain the fault assumption of passive faults in the channels, the active star couplers must not be given the capability to store frames and transmit them at a later time. They do not, however, explain why frame buffering should be prevented, nor do they explore the system-level implications of that constraint.

In the requirements for the central bus guardian Bauer *et al.* [3.2] specify the minimum number of bits that must be buffered in order to perform semantic analysis of frames. Semantic analysis is used to stop masquerading during startup and problems in integration due to transmission of frames with invalid C-state. This minimum buffer size is proportional to the number of bits in the longest frame, as well as the difference in clock rates between the nodes and the star coupler. No analysis of the effects of these restrictions on other system parameters, such as frame length and clock rates, is given.

Our objective in this research is to examine the engineering tradeoffs of adding centralized authority to a decentralized system by modeling a system based on the TTA with star topology and central bus guardians. We demonstrate why certain restrictions on the central authority are necessary by assessing the impact on fault-tolerance of full-frame buffering. We also analyze how added centralized authority can limit other system properties by showing that limiting buffer length also restricts maximum frame size and clock rates.

3.3. TTA Star Topology Model

In order to demonstrate why a central buffer should not be allowed to buffer an entire frame, we compare the fault-handling capabilities of TTA systems with frame buffering at the central bus guardian with systems containing star couplers with less authority.

3.3.1. Star-coupler Feature Sets

The four types of star-coupler authority we model are:

- Passive:

  does not stop frames

  does not shift frames in time
- Time windows:

  can open/close bus write access to nodes

  does not shift frames in time
- Small shifting:

same authority as *time windows*

also can make slight adjustment to frame timing (*e.g.*, shift slightly ahead to fit window)

- Full shifting:

  same authority as *small shifting*

  also can buffer frames to make large adjustment to frame timing (*e.g.*, save frames to send out at a later time)

3.3.2. The Formal Model

In order to investigate the behavior of TTP/C in the presence of central star coupler faults, we create a synchronous model of the channel, the couplers, and the nodes. Formally, we define a (finite) set of states S, a set of initial states I, and a transition relation R. The transition relation relates two states x, x'   S if and only if there is a transition from x to x'.

In order to obtain a tractable model, we abstract the behavior by merging all transitions that correspond to a single time slot in the TDMA schedule, *i.e.*, one transition of our model corresponds exactly to one TDMA slot. As the slots may have different lengths, a transition in our model may correspond to different interval in real-time, depending on the particular slot.

The set of states S is the set of possible valuations of the state variables. The state variables consist of the variables the nodes use to store their states and the variables used for the two star couplers. SMV allows specifying I and R using a list of constraints for each. The constraints are conjuncted to form I and R, respectively.

3.3.3. Modeling a Node

We model the following parts of the state of a single TTP/C node:

- The TTP/C standard describes a state machine with nine states that governs the behavior of the node: freeze, init, listen, cold_start, active, passive, test, await, and download. Initially, all nodes are in the freeze state. The variable used to store this state is denoted by `state`.
- We record the number of correct and bad frames received by the node during a TDMA round. This data is for the benefit of the clique avoidance algorithm. The names of the variables are `agreed_slots_counter` and `failed_ slots_counter`, respectively.

- Each node has a variable `slot` containing the current slot number in the TDMA schedule.
- In order to implement the "big bang" cold start algorithm, each node stores a flag `big_bang` that is set if a cold start frame is seen while the node is in the listen state.
- The listen state requires a timeout counter, which is modeled using a state variable for each node. In order to simplify the model, we count the number of TDMA slots using the variable `listen_ timeout`.

All other parts of the state of a node are not modeled. This includes, in particular, the application data.

We define the following constraints on the transition relation R. The unprimed variables denote the value of the state variable before the transition, the primed variables denote the value of the state variable after the transition. Furthermore, let `slots` denote the number of slots in the TDMA schedule.

3.3.3.1. FREEZE and INIT. From the freeze state, the node may make a transition into one of the states init, await, or test. From init, it may transition back into the freeze state or may proceed to the listen state. We model these choices nondeterministically.

```
state=freeze    state'    { freeze, init, await, test }
state=init    state'    { freeze, initialize, listen }
```

No further constraints are imposed on the variables if the node is in the freeze or init state.

3.3.3.2. LISTEN. In the listen state, the controller watches the channels for frames to integrate on. The controller may integrate on frames with explicit C-state or on cold start frames. Let the frame type on channel 0 be denoted by `channel0_frame`, and the frame on channel 1 be denoted by `channel1_frame`. Each can be one of `none`, denoting silence, `cold_start`, denoting a cold start frame, `c_state`, denoting a frame with explicit C-state, `bad_frame`, denoting a bad frame, or `other`, denoting a regular frame without explicit C-state.

As described above, nodes do not integrate on the first cold start frame received, but on the second. The variable `big_bang` is used in order to distinguish the first and second cold start frame.

```
big_bang'=
  if state listen then false
  else if big_bang then true
  else if channel0_frame=cold_start   channel1_frame=cold_start then true
  else false
  endif
```

Let `integrating_on_cold_start` denote a shorthand for the condition for integrating on the cold start frame currently on the channel:

```
integrating_on_cold_start=
  state=listen   (channel0_frame=cold_start   channel1_frame=cold_start)
      big_bang
```

In contrast to cold start frames, frames with explicit C state are used for immediate integration. Let `integrating_on_C_state` denote a shorthand for the condition for integrating on a C-state frame that is currently on the channel:

```
integrating_on_C_state=
  state=listen   (channel0_frame=c_state   channel1_frame=c_state)
```

The node integrates if either condition is true:

```
integrating=integrating_on_C_state   integrating_on_cold_start
```

If the node is integrating, the time slot counter is set to the value found on the bus plus one. This value is denoted by `id_on_bus`. Thus,

```
(state=listen   integrating)    slot'=
  if id_on_bus=slots then 1
  else id_on_bus+1
  endif
```

For startup, the node maintains a timeout counter for the listen state. This counter is initialized with the number of slots plus the number of the slot that is assigned to the node. This slot is denoted by `node_id`. The timeout is reset also if a good frame is seen on the channels. If no correct frame is received, the node counts down the listen timeout counter.

```
listen_timeout'=
  if (state listen   state'=listen)   channel0_frame=cold_start
    channel1_frame=cold_start   channel0_frame=other
    channel1_frame=other then node_id+N
  else if listen_timeout 0 then listen_timeout-1
  else 0
  endif
```

If the node is integrating, the node transitions into the passive state. In case of a timeout, it transitions into the `cold_start` state. If there is a cold start frame on either channel that is not used for integration, the node stays in the listen state even if the timeout counter just reached zero.

```
state=listen    state’=
  if integrating then passive
  else if channel0_frame=cold_start   channel1_frame=cold_start then
    listen
  else if listen_timeout=0 then cold_start
  else listen
  endif
```

3.3.3.3. COLD START. Upon entering the cold start state, the node initializes the slot counter with its own slot number.

```
(state cold_start   state’=cold_start)    slot’=node_id
```

As a shorthand, let `next_slot` denote the number of the next TDMA slot. This is `slot+1` if `slot < slots`, and 1 otherwise. During cold start, the node maintains the slot counter as if integrated.

```
(state=cold_start   state’=cold_start)    slot’=next_slot
(state=cold_start   state’=active)        slot’=next_slot
```

During cold start, the node monitors the frames on the buses. Once one TDMA round is finished, and traffic is observed, the node performs the clique avoidance test. If it succeeds, it transitions into the active state. Otherwise, it transitions back into the listen state.

```
(state=cold_start)    state’=
  if next_slot=node_id then
    if agreed_slots_counter’<=1   failed_slots_counter’=0 then cold_start
    else if agreed_slots_counter > failed_slots_counter then active
    else listen
    endif
  else cold_start
  endif
```

3.3.3.4. ACTIVE. In the active state, the controller maintains the slot counter:

```
(state=active   state'=active)     slot'=next_slot
(state=active   state'=passive)    slot'=next_slot
```

It may nondeterministically transition to freeze or passive:

```
state=active    state'    { freeze, active, passive }
```

Let `frame_sent` denote a shorthand for the frame that is sent by the node:

```
frame_sent=
  if state=active   slot=node_id then c_state
  else if state=cold_start   slot=node_id then cold_start
  else none
  endif
```

3.3.3.5. The Star Couplers. Each of the two couplers may have one of the following error states: `none`, corresponding to error-free operation, `silence`, which replaces any frame that is sent on the channel belonging to the coupler by silence, `bad_frame`, which places a bad frame or noise on the bus, regardless if a frame was sent or not. Furthermore, the `out_of_slot` fault corresponds to re-sending the last frame received by the coupler.

The fault state of the coupler is denoted by `fault`, *i.e.*, the full names of the variables are `coupler0.fault` and `coupler1.fault`, respectively. The `out_of_slot` fault occurs only if the couplers are configured for full time shifting. All other faults may be caused by any configuration. In accordance with the TTP/C fault hypothesis, we require that at most one coupler has a fault at a given time:

```
couplerA.fault=none   couplerB.fault=none
```

In order to model the behavior of the bus system in case of an out-of-slot error, we add a state variable for each coupler containing the id and type of the frame that was received last. The variable for recording the id is called `buffered_id`, the variable for the type is called `buffered_frame`. The variables are initialized with `0` and `none` respectively.

Let `channel_id` denote the id on the channel belonging to the coupler. Let `channel_frame` denote the frame type. Then,

```
buffered_id'=
```

```
    if channel_id=0 then buffered_id
    else channel_id
    endif
  buffered_frame'=
    if channel_id=0 then buffered_frame
    else channel_frame
    endif
```

The frame currently on the channel is

```
  channel_frame=
    if fault=silence then none
    else if fault=bad_frame then bad_frame
    else if fault=out_of_slot then buffered_frame
    else if node[i].node_is_sending then node[i].frame_sent
    else none
    endif
```

The id on the channel is defined analogously.

3.4. Experimental Results

3.4.1. Property Checked

We use the following correctness criterion: As the nodes are modeled not to fail, no single fault may prevent any node from integrating or losing membership. The TTP/C standard requires that the affected node makes a transition into the freeze state in this situation, *i.e.*, we check that

```
 (state=active   state=passive)    state' freeze
```

holds on all reachable states.

3.4.2. Results

For the passive, time windows, and small shifting couplers we verify that the property above holds. For the configuration that allows any star coupler to buffer full frames and replay them in a later time slot, we obtain counter examples from the model checker that demonstrate a failure.

If a property does not hold, the SMV model checker produces a trace from any initial state to a state in which the property does not hold. SMV produces the shortest possible trace. However, the shortest error trace contains four out-of-slot errors. As one might argue that such an accumulation of errors is unlikely, we add a constraint to the model which limits the number of out-of-slot errors to one.

This results in a slightly longer trace, but still produces an error that is caused by a duplicated cold start frame:

1) Initially, all nodes are in the freeze state.
2) In the next state, all the nodes transition into the init state.
3) Node A makes a transition into the listen state. The other nodes remain in the init state.
4) The listen timeout counter of node A decreases down to zero. Node B finishes its initialization and transitions into the listen state.
5) Node A sends a cold start frame on the bus. Node B ignores the frame due to the "big bang" requirements. The nodes C and D make a transition into the listen state.
6) A faulty star coupler replays the previous cold start frame. Node B integrates on it, in compliance with the "big bang" requirements.
7) Node A sends another cold start frame. Nodes C and D use this frame to integrate on. Then transition into the passive state.
8) Node C sends a C-state frame. Node B considers this frame a faulty frame.
9) Node D sends a C-state frame. Node B considers this frame a faulty frame.
10) Node B freezes due to a clique avoidance error.

The error may also be triggered by duplicating a C-state frame. We obtain such a trace by adding a constraint which prohibits the duplication of cold start frames:

1) Initially, all nodes are in the freeze state.
2) In the next state, all the nodes transition into the init state.

3) Node A makes a transition into the listen state. The other nodes remain in the init state.

4) The listen timeout counter of node A decreases down to zero. Node C finishes its initialization and transitions into the listen state.

5) Node A sends a cold start frame on the bus. Node C ignores the frame due to the "big bang" requirements. Node B makes a transition into the listen state.

6) Node A sends another cold start frame. Node C integrates on the frame, and transitions into the passive state. Node B ignores it according to the "big bang" requirements. Node D makes a transition into the listen state.

7) A faulty star coupler replicates the previous frame into the next slot. Node D integrates on it, making a transition into the passive state. The other nodes consider the frame to be faulty.

8) In the next three frames, the nodes A, B, and C each send a C-state frame. Node D considers all to be a faulty frames.

9) Node D freezes due to a clique avoidance error.

Both traces are generated in less a than a minute on a 1.5 GHz AMD machine.

3.5. Analysis

The results from our model show that faults that cause frames to be transmitted outside of their assigned time slot can lead to failures during the startup and integration phases of protocol service. Central bus guardians that are allowed to buffer an entire frame are susceptible to these types of faults, even in systems with independent, redundant central guardians. If one bus guardian becomes faulty and sends frames in the wrong time slot and the other bus guardian is behaving correctly, receiving nodes that have already integrated into the cluster will recognize that the frame on the channel with the faulty bus guardian is incorrect because the frame's C-state (which contains the incorrect global time and slot position) does not match its own C-state. These nodes will use the correct frame on the non-faulty channel, or no frame at all if none is transmitted in that time slot. However, nodes that are integrating, either during a cold-start or into a running cluster, are not able to determine that the frame is incorrect, and may use the faulty frame.

It is important to show the problems with system dependability that arise when a central guardian is allowed to buffer an entire frame because there are several reasons why a system architect might be tempted to buffer an entire frame. One reason is cost and ease of implementation. A central guardian could use the same controller as the nodes to receive frames, buffer them, and send them out again on the network. This design would also violate the requirement that the central guardian be prohibited from knowing how to generate a frame (including a valid CRC); however, a central guardian that just receives frames and sends them out again (without re-creating them) would similarly be a simple solution that requires buffering. The addition of new functionality to the central guardian could also lead to buffering of frames. For example, an active central guardian that keeps "mailboxes" with recent data values could help provide data continuity if frames are corrupted by providing slightly stale values instead of no value. A central guardian could also provide prioritized message service (*e.g.*, CAN emulation) if it were allowed to buffer frames and send them in a specially reserved time slice, in priority order. Both of these enhanced functions would require buffering full frames.

In addition to examining the importance of restricting buffer size, it is also useful to analyze the effects of this restriction on other system properties. For example, in the TTA with star topology and central bus guardians, active signal reshaping (in the time and value domains) and semantic analysis of the C-state require the guardian to buffer a minimum number of bits of a frame [3.2], which is determined from the maximum frame length and the relative difference in rates between the guardian and the nodes.

The restriction on the maximum buffer size (less than the smallest frame) corresponds to a restriction on the relationship between frame lengths and clock rates. If the clock rates of the sending node and the central guardian are different, the central guardian must buffer some part of a frame before it can begin forwarding it. If the central guardian is faster, it must wait to send the frame to ensure that it does not run out of bits to send during the transmission. If the central guardian is slower, it must buffer some of the bits until it has time to send them. This corresponds to the idea of a “leaky bucket” where the fill rate is not equal to the drain rate. The minimum buffer size $B_{\min}$ is given by:

$$B_{\min} = {}_{\mathrm{le}} + \quad * f_{\max} \tag{3.1}$$

where ${}_{\mathrm{le}}$ is the number of bits required for line encoding, and $f_{\max}$ is the maximum frame size. is the relative difference in clock rates of the faster ${}_{\max}$ and the slower ${}_{\min}$:

$$= (\quad {}_{\max} - \quad {}_{\min}) / \quad {}_{\max} \tag{3.2}$$

Bauer *et al.* [3.2] find that the $* f_{\max}$ term was multiplied by a factor of 2, however the assumptions in the chapter that lead to that conclusion are unclear. Therefore, we use equation (3.1) for this analysis.

It has been shown that a central bus guardian must be prohibited from buffering an entire frame; therefore, the maximum buffer size B_{max} is limited by the smallest frame transmitted on the network, such that:

$$B_{max} = f_{min} - 1 \tag{3.3}$$

where f_{min} is the number of bits in the smallest frame. If b_{min}, $_{le}$, and are known the largest allowable frame, f_{max}, is found by setting $B_{min} = B_{max}$:

$$f_{max} = (f_{min} - 1 - {}_{le}) / \tag{3.4}$$

The maximum frame size, therefore, is inversely proportional to the relative difference in clock rates between the nodes and guardian. Suppose the nominal clock rates of the star coupler and all of the nodes are equal. Even in this case, variations in the manufacturing process could lead to slight variations in the actual clock rates of the nodes and, although very small, will not be zero. For example, a typical commodity crystal oscillator may have a clock rate that varies by approximately 100ppm. If we assume that in the worst-case scenario the clock in the star coupler is 100ppm fast and the node is 100ppm slow, the difference in clock rates between the two is:

$$= 2 * (0.0001) = 0.0002 \tag{3.5}$$

According to the TTP/C Bus-Compatibility Specification [3.12], the shortest frame in TTP/C, an N-frame with no application data and an implicit CRC, is 28 bits long (4 bits for the mode change request and frame type and 24 bits for the CRC). The minimum cold-start frame requires 40 bits (1 bit for the frame type, 16 bits for the global time, 9 bits for the round-slot position, and 24 bits for the CRC)[3.5, 3.12]. The minimum frame with explicit C-state is an I-frame with 48 bits (4 for the mode change request and frame type, 16 bits for the global time, 16 bits for the MEDL position, 16 bits for membership, and 24 bits for the CRC) [3.12]. Although our model showed that only cold-start frames and frames with explicit C-state caused failures when an out-of-slot fault occurred, the system should be designed to prevent any frame from being transmitted outside of its assigned time slot; therefore, this analysis assumes a maximum buffer length that is less than 28 bits long.

Using the values $f_{min} = 28$, $= 0.0002$, and $_{le} = 4$ we get:

$$f_{max} = (28 - 1 - 4) / (0.0002) = 115{,}000 \text{ bits} \tag{3.6}$$

The longest allowable frame size for TTP/C is 2076 bits (an X-frame with 4 bits for mode change request and frame type, 96 bits for C-state, 1920 data bits, 48 bits for two CRCs, and 8 bits for CRC padding) [3.12]. In this example longest allowable frame size of 115,000 bits is much larger than the number of bits in the largest allowable frame.

This calculation is for nodes and central bus guardians with the same nominal clock rate. What happens if the nominal clock rates of the nodes and the central guardian are not equal? It is possible to calculate the maximum allowable difference in clock rates, given set values of f_{min} and f_{max}, by rearranging equation (3.3):

$$= (f_{min} - 1 - \ _{le}) / (f_{max}) \tag{3.7}$$

The smallest possible value of f_{max} is the size of the largest frame required for protocol operation. This is an I-frame of length 76 bits (4 bits for the mode change request and frame type, 16 bits each for global time, MEDL position, and membership, and 24 bits for CRC). If we set f_{max} = 76 bits, $_{le}$ = 4,and f_{min} = 28 we get:

$$= (28 - 1 - 4) / (76) = 0.3026 \tag{3.8}$$

This means that in order to achieve minimal protocol operation, the relative difference in clock rates between the bus guardian and the slowest node can not be more than 30.26%.

Now suppose nodes are allowed to send X-frames of maximum length, the longest frames in the protocol. With f_{max} = 2076, the limit of becomes:

$$= (28 - 1 - 4) / (2076) = 0.0111 \tag{3.9}$$

In this scenario, a system that utilizes the maximum possible frame size cannot have a relative difference in clock rates between the bus guardian and the slowest node that is greater than 1.11%. The situation becomes more constrained if the protocol is altered to allow longer frames, or if the equation in [3.2] is used.

This dependancy between clock rates and frame lengths is not limited to the TTA with a star topology and central bus guardian, but rather, it is a fundamental property of any decentralized system with a centralized supervision of communication that is prohibited from buffering complete frames. Whenever the clock rate of the central supervisor differs from the clock rate of any one of the supervised components, this central supervisor must buffer some part of the communication and the minimum size of this buffer B_{min} is proportional to the difference in clock and the maximum frame size f_{max}. In order to maintain system dependability, the maximum size of this buffer B_{max} must be less than the minimum frame size f_{min}. In other words if the difference between the minimum and maximum buffer is large, the difference in clock between the central authority and the guarded nodes cannot be. From equations (2) and (7) we can calculate the ratio of $_{max}$ / $_{min}$:

$$_{max} \ / \ _{min} = (f_{max}) / (f_{max} - f_{min} + 1 + \ _{le}) \tag{3.10}$$

Figure 3.3 depicts this relationship for $_{le} = 4$, where the range of possible values lie below the curve. This graph shows that systems with a wide range of frame lengths cannot also have a wide range of clock rates. The same is true for systems with a short maximum frame size. It is interesting to note that if the maximum and minimum frame size are both 128 bits the ratio of $_{max}$ / $_{min}$ is not equal to f_{max}, but rather is $f_{max} / 5 = 25$. This is due to the 1 + $_{le}$ term in equation (3.10). This is a significant limit at high clock ratios when the transmission time of a frame at the high clock speed approaches the $_{le}$ time at the low clock speed.

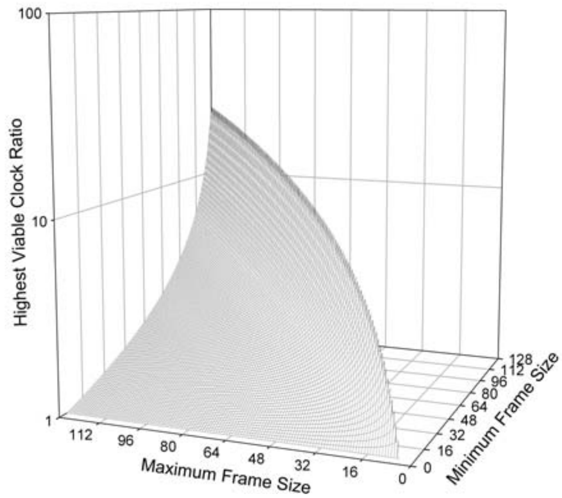

Figure 3.3: Relationship between frame size range and ratio of clock rates

One simple way to comply with this restriction is to simply require all nodes and bus guardians to have the same clock rate. However, there are several reasons why a system architect might choose to have some nodes have different connection speeds to the hub. Slow, inexpensive nodes (that need a few short frames) might use slow, inexpensive links, whereas fast, capable nodes (that usually generate and consume more network traffic) might have fast links. Different clock rates may also be desirable if the hub is designed to filter traffic. From a TDMA point of view, each frame could get the same size time slice. Frames that are readable to slow nodes would have only a few bits. Frames with more bits would have to run at a higher bit rate, and would only be readable by faster, more capable nodes. An active central guardian could allow faster nodes to exchange longer frames, while at the same time preventing these unreadable frames from being sent to the slower nodes. Unfortunately, it may not be possible to safely implement these systems because the restrictions on buffer size might not allow significant differences in clock rates.

3.6. Conclusion

We have demonstrated that allocating some types of authority to the central bus guardian can lead to the same failures the authority was intended to prevent. In particular, systems that are allowed to stop transmission of frames with invalid C-states or invalid cold-start frames and have the authority to buffer entire frames can experience the same problems in startup and re-integration when faults occur in a star coupler as systems without centralized control that experience faults at individual nodes.

The example presented in this chapter focused on a synchronous system, the TTA. However, the same effect could also be observed in asynchronous systems. The failures described above were essentially masquerading failures caused when the centralized authority introduced invalid frames containing erroneous identification information onto the network. The same type of masquerading failures could occur in a distributed, asynchronous system because the underlying issue is not timing, but rather identification. A central authority with access to the other nodes' knowledge (e.g., identification methods) may have the ability to introduce masquerading failures into a decentralized system, whether that system is synchronous or asynchronous.

In addition, we have shown that restrictions on the authority of a central buffer correlate to increased restrictions on other system parameters. In the case of the TTA with a star topology and central bus guardians, the restrictions on buffer size in the star coupler lead to increased restrictions on frame size and clock rates. In this example, the restrictions are primarily due to the fact that frame senders are partially identified by the frame timing. The problem is not limited to synchronous systems, however, because some of the failures occurred with cold-start frames, which are used in normal operation before the system has synchronized .

System architects may be tempted to add centralized authority to a decentralized system in order to increase the efficiency, dependability, or functionality of the system. Our results show that this additional authority must be scrutinized to ensure that it does not negatively affect other properties of the system.

3.7. References

[3.1] P. Verissimo and L. Rodrigues, *Distributed Systems for System Architects*, Kluwer Academic Publishers, 2001.

[3.2] G. Bauer, H. Kopetz, and W. Steiner, "The Central Guardian Approach to Enforce Fault Isolation in the Time-Triggered Architecture," in *Proceedings of the Sixth IEEE International Symposium on Autonomous Decentralized Systems (ISADS'03)*, Pisa, Italy, April 2003, pp. 37-44.

[3.3] A. Ademaj, "Slightly-Off-Specification Failures in the Time-Triggered Architecture", in *Proceedings of the Seventh IEEE International Workshop on High Level Design Validation and Test (HLDVT'02)*, Cannes, France, October 2002, pp 7-12.

[3.4] Model Checking at Carnegie Mellon University http://www-2.cs.cmu.edu/~modelcheck/

[3.5] TTTech Computertechnik AG, *Time-Triggered Protocol TTP/C High-Level Specification Document edition 1.0.0*, TTTech Computertechnik AG, Vienna, Austria, 2002.

[3.6] L. Lamport, R. Shostak, M. Pease, "The Byzantine Generals Problem," in *ACM Transactions on Programming Languages and Systems*, vol. 4 no. 3, pp. 382-401, 1982.

[3.7] A. Ademaj, H. Sivencrona, G. Bauer, J. Torin, "Evaluation of Fault Handling of the Time-Triggered Architecture with Bus and Star Topology," in *Proceedingsof the 2003 IEEE International Conference on Dependable Systems and Networks* (DSN'03), San Francisco, California, June 2003, pp. 123-132.

[3.8] H. Pfeifer, D. Schwier, and F. v. Henke, "Formal Verification for Time-Triggered Clock Synchronization," In *Proceedings of Seventh IFIP International Working Conference on Dependable Computing for Critical Applications*, pp. 207-226, 1999.

[3.9] J. Rushby, "Systematic Formal Verification for Fault-Tolerant Time-Triggered Algorithms" *IEEE Transactions on Software Engineering*, vol. 25, no. 5,pp. 651-660, 1999.

[3.10] H. Pfeifer, "Formal Verification of the TTP Group Membership Algorithm". In *IFIP TC6/WG6.1 International Conference on Formal Description Techniques for Distributed Systems and Communication protocols and Protocol Specification, Testing and Verification, FORTE/PSTV 2000*, Pisa, Italy, October 2000, pp. 3-18.

[3.11] G. Bauer, H. Kopetz and P. Puschner, "Assumption Coverage under Different Failure Modes in the Time-Triggered Architecture", In *Proceedings 7th IEEE International Conference on Emerging Technologies and Factory Automation*, Antibes - Juan les Pins, France, October 2001, pp. 333-341.

[3.12] TTTech Computertechnik AG, *Time-Triggered Protocol TTP/C Bus-Compatibility Specification edition 1.0.0*, TTTech Computertechnik AG, Vienna, Austria, 2002.