Adaptive Fault Tolerance for Spacecraft[1]

Myron Hecht
SoHaR Incorporated
8421 Wilshire Blvd., Suite 201
Beverly Hills, CA 90211
323-653-4717
myron@sohar.com

Herbert Hecht
SoHaR Incorporated
8421 Wilshire Blvd., Suite 201
Beverly Hills, CA 90211
323-653-4717
herb@sohar.com

Eltefaat Shokri
Compaq Computer
19333 Vallco Parkway
Cupertino, CA 95014

Eltefaat.Shokri@compaq.com

Abstract— This paper describes the design and implementation of a software infrastructure for real-time fault tolerance for applications on long duration deep space missions. The infrastructure has advanced capabilities for Adaptive Fault Tolerance (AFT), i.e., the ability to change the recovery strategy based on the failure history, available resources, and the operating environment. The AFT technology can accommodate adaptive or fixed recovery strategies. Adaptive fault tolerance allows the recovery strategy to be changed on the basis of the mission phase, failure history, and environment. For example, during a phase where power consumption must be minimized, there would be only one processor in operation. Thus, the recovery strategy would be to restart and retry. On the other hand, if the mission phase were in a time-critical mode (e.g., orbital insertion, encounter, etc.), then, multiple processors would be running, and the recovery strategy would be to switch from a leader copy to a follower copy of the control software. In a fixed recovery strategy, there is a specified redundant resource which is committed when certain failure conditions occur. The most obvious example of a fixed recovery strategy is to switch over to the standby processor in the event of any failure of the active processor.

TABLE OF CONTENTS

1. INTRODUCTION

Spaceborne fault tolerance capabilities are necessary to maximize mission success probabilities [Alk93]. Over extended missions, it is likely that failures will occur due to design deficiencies interacting with these environmental conditions. This is demonstrated by Table 1-1, which shows the distribution of failures causes for past spacecraft systems [Hec88].

About 56% of failures are of design and/or environmental causes. This proportion strongly suggests that environmental changes for autonomous spacecraft are major concerns in designing fault-tolerance capabilities. The study also showed that about 22% of all reported failures are of transient nature. Thus, means of avoiding or masking these types of failures are also necessary.

Table 1-1 . Distribution of Spacecraft Failures

Cause	Definition	Percent
Design	(i) selection of parts that do not possess sufficient strength and (ii) selected parts do not allow for the full range of spacecraft operation.	30.1
Environ-ment	unanticipated effects are encountered or when the magnitude of anticipated events was greater than specified or expected.	26.5
Parts	associated with random failures in parts which cannot be related to the quality of the parts.	15.5
Quality	assigned as the cause of failures where there are repeated failures in the same part or assembly that cannot be attributed to design or environment.	5.0
Oper-ation	failures caused by sending improper commands to the spacecraft or faulty ground software.	5.9
Other Known	Probably due to parts. The reason that places a failure in this category is insufficient data in the report.	3.3
Un-known	The class of other knowns includes, for example, early depletion of the consumables (such as the attitude control gas).	13.3

The results of Table 1-1 reflect primarily spacecraft hardware. However, the size of software employed in spacecraft systems has grown about 2000% since 1970 [Sta94]. This growth increases the likelihood of software faults leading to mission failures. The results of research on the Space Shuttle's similar large safety-critical systems show that a careful development and an intensive testing process can significantly reduce permanent design faults in both hardware and software. However even such software fails, if for rare combinations of conditions which have been overlooked in the testing phase [Hec93].

Thus, fault tolerance provisions must cover Hardware Physical Faults, Rare Conditions in Software, and

[1] 0-7803-5846-5/00/$10.00 © 2000 IEEE

Overlooked/Unusual Environmental Effects. However, Spacecraft for Deep Space Missions, such as Pluto-Kuiper Express, have severe resource constraints due to limited power, weight, and size [Bro96, Bar99]. Thus, although internal fault tolerance capabilities are necessary, the need for redundant resources that are needed for fault tolerance must also be minimized. For example, although during most of the mission, response times are not critical, certain mission phases, such as flyby or orbital insertion, are extremely time critical and failures to meet processing deadlines during these mission phases may result in a significant loss of science data — and a corresponding degradation in the value of the overall mission. In order to address both the need to minimize on-board redundant resources and the requirement for reliability during deep space missions, a flexible approach is necessary. Adaptive fault tolerance [Law95] is a mechanism which allows such flexibility. The research described in this report enables a flexible recovery mechanism based on available resources, environmental demands and conditions, and failure history.

2 ADAPTIVE FAULT-TOLERANCE CONCEPTS

This section discusses the basic concepts of adaptive fault tolerance. The first subsection identifies the essential features of an effective adaptive fault-tolerance mechanism. The second discusses additional characteristics of an adaptive fault-tolerance mechanism suited for dynamic reconfigurable systems. Finally, the candidate middleware architecture for an efficient implementation of the adaptive fault-tolerance mechanisms with identified core characteristics are presented.

Goals of adaptive fault-tolerance mechanisms

The purpose of adaptive fault-tolerance is to enhance system reliability, performance, and survivability through the following qualities [Sho97, Sho98]:

 Effectiveness: An adaptive fault-tolerance (AFT) approach should significantly improve reliability (over a single string system). The overhead incurred by the adaptive subsystem must be within acceptable limits.

 Enhanced Resource Utilization: The goal of the reactive adaption mechanism is minimizing resource utilization, while retaining required reliability requirements. AFT should enable the utilization of processing and other resources such that temporary deactivation (i.e., powering down) or alternate use of redundant channels is possible.

 Generality: The AFT mechanisms must be sufficiently general to handle a variety of credible fault classes. Specifically, the AFT mechanisms should enable system designers to handle sensor, processor hardware, system software, and application software failures. The mechanisms should be able to handle both transient and permanent failures. The AFT middleware must be able to handle both hardware and software failures.

 Appropriate response: The AFT mechanisms must respond to changes in the environment, mission phase, system state, and user profiles within an acceptable time-period. Table 2-1 shows examples of such changes.

Table 2-1 . Potential Changes to which adaptation profiles must respond

Class	Examples
Environ-mental	Temperature cycling leading to a higher failure rate of hardware components and interfaces. High radiation zones leading to temporary increases of transient failures in processing.
Mission Phase	Changes in response time (e.g., cruise vs. orbital insertion or flyby). Capacity surges due to scientific instruments during or after an encounter.
System state	Power capacity. Processor configuration changes. Sensor failures or actuator failures. Transient or permanent processor hardware failures.
Use Profiles	Reprogramming and different uses of scientific instruments. Increasing mission length.

 Configurability: The user and/or the developer of an application may provide application-specific parameters that affect the adaptation decisions. An effective AFT manager should be able to accept these parameters during the execution of the application and factor them into the decision-making process.

 Versatility: A suitable adaptation policy should consider various strategies (or modes) of adapting the system in response to anomalous behaviors (actual or anticipated). The adaptation can be envisioned, for example, in one of the following ways: (i) switching from one fault-handling mode to another, (ii) modifying parameters of the currently-used fault-handling mode, or (iii) modifying service attributes.

 Simple Interfacing: The AFT Middleware should be implemented as separate layers and must have a simple and clear interface with the application, in order to maximize its openness and reusability. The application should then interact with the middlewarelayer only through this interface.

Figure 2-1 illustrates a candidate set of input parameters provided by the user and/or the run-time systems. As depicted in the diagram, two sets of inputs are provided to the adaptation decision-maker. The application designer can supply the application-specific inputs, such as application performance/reliability attributes and preferred transition policies. During system operation, other system components (which are collecting information regarding the system behavior and anomalies) can provide the adaptation mechanism with the changes in the system, as well as the environment. The adaptation mechanism will be activated in an appropriate time and decide on the most suitable

execution modes for application tasks. In order to enforce the execution modes selected by the adaptation decision-maker, the system may reallocate available system resources.

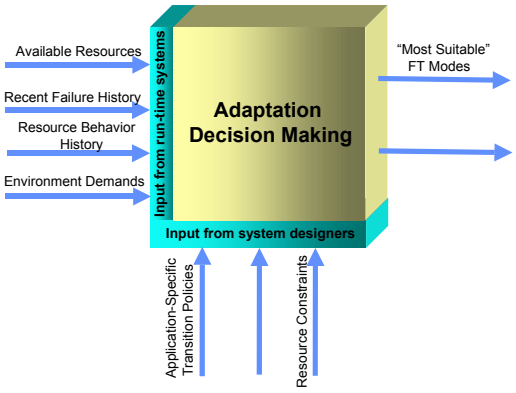

Figure 2-1. Adaptation Decision Making: Sample Input/Output

Fault-tolerance middleware architecture

To maximize the reusability, the adaptive fault-tolerance subsystem is implemented as a separate middleware layer denoted as the adaptive fault-tolerance manager (AFTM), such that the application designer can utilize the provided services in a well-defined and easy-to-use manner. The overall architecture of the AFTM is shown in Figure 2-2.

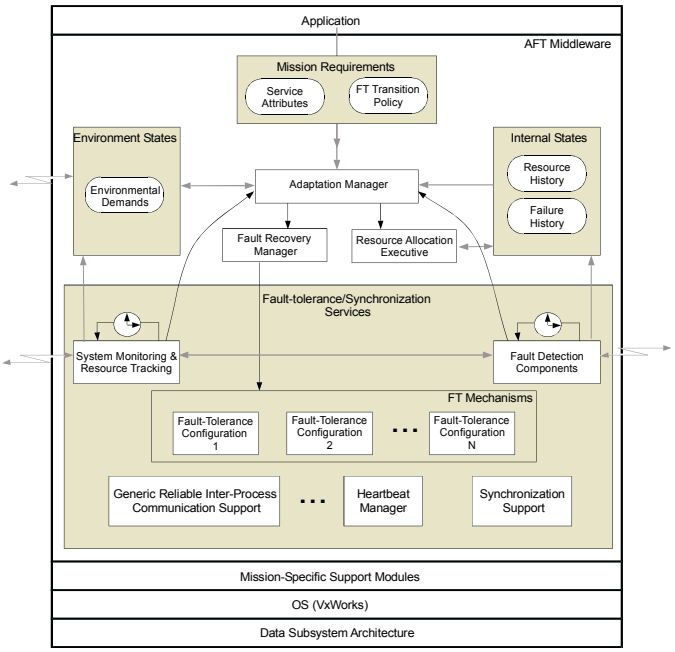

Figure 2-2. Adaptive Middleware Components

As shown in the figure, the middleware is built on top of a Generic Communication Layer which encapsulates platform/protocol dependent aspects of the underlying communication subsystem and thereby provides the middleware with a generic set of inter-node communication services. The middleware depends on time services of the operating system. Modern commercial operating systems, such as Solaris and Windows NT, may provide the minimum requirements providing that the system operates in an isolated inter-network environment. The adaptation mechanism employed by the AFTM utilizes the following three databases:

- Environmental Database: This database maintains a representation of the environment in which the system operates, including any recent changes in environmental conditions or demands that should be taken into consideration by the AFTM. Updates to the database are made by sensors interfacing with the environment and by AFT Middleware components responsible for monitoring and diagnosis (such as the Application Execution Monitoring component) based on the recent diagnosis knowledge.

- Internal State Database: This database maintains the recent failure history as well as the history of recent behavior of system resources. The Network Reconfiguration Manager (NRM) and the Resource Allocation Executive (RAE) update this database when an anomalous behavior of a system component is diagnosed.

- User Requirement Database: The user informs the AFTM about the specific characteristics and requirements of the application by providing (1) application-specific adaptation policies (if they exist), and (2) application attributes such as acceptable reliability and performance characteristics of each application task.

The Adaptation Manager (AM) component selects the most suitable fault-handling and resource-allocation modes of the system based on the current contents of these three databases. The adaptation decision made by AM is forwarded to the Resource Allocation Executive component to enforce the changes by reallocating available resources.

Any change in the health status of the software and hardware components of the cooperating distributed AFT Middleware, as well as the status of the application components, must be made known by the Adaptation Manager within an acceptable time period (i.e., a minimal delay). The Network Reconfiguration Manager (NRM) component is responsible for the fast detection of any anomalous behavior of software and/or hardware components of the application, or AFT Middleware. NRM also reconfigures the network when a permanent malfunction of a resource is diagnosed.

The Application Execution Monitoring (AEM) component provides the user with the current status of application components. It is mainly responsible for monitoring the health status of active objects running in the nodes in the distributed system. This system monitoring facility enables

the user to selectively monitor the health-status of various objects and instruct AFT Middleware to take an appropriate action when the system enters an unexpected abnormal state.

The AFT Middleware provides a variety of fault-tolerant execution modes from which the most suitable execution mode can be selected for each application task based on the task requirements and resource availability.

The primary mechanism is application-level fault-tolerance, reliable communication through logical channels (see Chapter 4). AFT currently uses lower level communication mechanisms, such as socket-based peer-to-peer communication, in order to take advantages of the optimization that can be done in lower-level abstracts. However, the middleware can easily ported to OMG-CORBA (Common Object Request Broker Architecture) compliant Object Request Brokers (ORRs). ORB specifications and allow the specifying the application quality-of-services, such as the periodicity of the methods and their associated deadlines, and maximum-execution-times. It supports both dynamic and static scheduling. The drawback or ORBs is demanding memory and processor requirements.

3. ADAPTATION POLICIES

This section discusses adaptation policies using a dual redundant processing architecture as an example. This architecture was chosen because dual redundancy is common on long duration spacecraft [Bro96]. Where the architecture consists of more than two processors, the approach can be extended through (a) distributed replicated pairs of applications among the operational group, or (b) using two operational processors with the remaining complement being unpowered spares. The first subsection describes the system architecture, the second discusses the software execution model, which facilitates validation of the adaptive mechanism. The third subsection identifies seven fault-tolerant execution modes of the engineering and science tasks and their transition conditions. The fourth and final subsection describes fault recovery mechanisms employed by the adaptive fault-tolerance mechanism.

The dual-redundant architecture consists of two identical processors (with local memories). The speed of each processor can be adjusted by software. Under nominal conditions, one processor is executing the engineering data task while the other one is executing the science data tasks. In the case of a permanent failure of a processor, the other processor detects the failure of its peer and takes over the peer's responsibilities. It doubles its speed and executes both engineering and science data tasks.

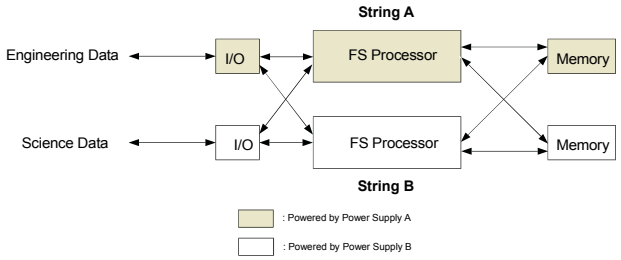

Figure 3-1. Dual Redundant System Architecture

It is assumed that the architecture is designed to contain the effects of the failure in the smallest unit possible. A common technique for containing a failure is to restrict the failure domain to a computing node (i.e., a processor and its memory). One way to achieve this is for the computing node to possess a capability to recognize the occurrence of failures and silence itself. There are two ways to construct a fail-silent processor; (i) using hardware redundancy, and (ii) using self-checking software.

The spacecraft application software consists of the engineering subsystem for performing spacecraft functions, such as navigation, and the science subsystem for spacecraft specific missions. The engineering, as well as science subsystem, software is composed of periodic and aperiodic tasks. Under this model, application tasks are categorized into the following classes:

- *Critical Periodic Tasks:* Normally, these tasks have very stringent completion deadline requirements, meaning that they should be activated and completed in specified time windows, even in the presence of faults or system overloads.

- *Critical Aperiodic Tasks:* Very few application tasks belong to this category. However, in exceptional conditions, such as occurrence of unexpected external events, the system's actions are of this type (i.e., critical for the system survival and aperiodic). The proposed execution model should have the capability of timely execution of these tasks.

- *Non-critical Periodic Task:* Non-critical periodic tasks are executed periodically. If the system delays their execution – or fails to execute them for a limited number of periods -- there will not be catastrophic consequences.

- *Non-critical Aperiodic Tasks:* Tasks such as sending non-critical information to the ground control are non-critical aperiodic tasks. These tasks can be delayed if the system is in an intensive transient overload situation.

Most engineering data tasks are critical. However, some science tasks are considered critical as well. It is assumed that during the system design, all critical aperiodic and periodic tasks are identified and their periods, deadlines,

and worst-case execution times are calculated. It is also assumed that scheduling analysis is performed to validate the feasibility of timely execution of all critical tasks even in the presence of undesired events, such as occurrence of failure and transient system overloads.

As shown in Figure 3-1, the engineering and science tasks are executed in two separate strings. As will be discussed in the next section, the adaptation mechanism identifies other execution modes so that the system can switch from one execution mode to another for a better utilization of resources.

The objective of the adaptive fault-tolerance mechanism is to match redundancy and resource consumption with the mission phases performance and reliability requirements. For example, during the cruise, the spacecraft is in a stable equilibrium and can tolerate long interruptions of services within the navigation, guidance, and control functions. During launch, orbital insertion, or flyby phases, even a short interruption may lead to a catastrophic consequence. Thus, the following adaptation strategy is adapted:

- In the launch and reentry phases, both strings A and B should execute the GNC function. This facilitates a very short fault-recovery delay.

- Self-checking execution of the GNC functions in the most reliable string seems to be sufficient during cruise, permitting the other string to be powered down and thus saving power.

To identify conceivable execution modes for spacecraft applications, a simple classification of inputs to the adaptation mechanism was used:

Recent Failure History: (i) recent failure rate is low (i.e., below a known threshold), (ii) recent failure rate is high (above the threshold)

Available Resources:: (i) none of the processors manifested a faulty behavior, or (ii) one processor is available, the other is either shut-down or being tested.

Required Timeliness / Tightness of Action Deadlines: One simple classification is: (i) tight deadlines are required, or (ii) loose deadlines are required.

Resource (Behavioral) History: (i) processor A seems more reliable than processor B (fewer failures in processor A), or (ii) processor B seems more reliable than processor A (fewer failures in processor B).

Figure 3-2 shows all possible input states for the adaptation mechanism and corresponding execution modes. Figure 3-3 organizes possible system modes according to their "resource availability" and "environmental conditions". Mode 7 has the most severe conditions in both available resources and environmental demand, such that timely and correct execution of both critical and non-critical tasks are not feasible. Hence, the system is asked to execute only the critical tasks until the system state is improved. In the other extreme, Mode 2 is the most relaxed state (all resources are available and environmental demands are not high).

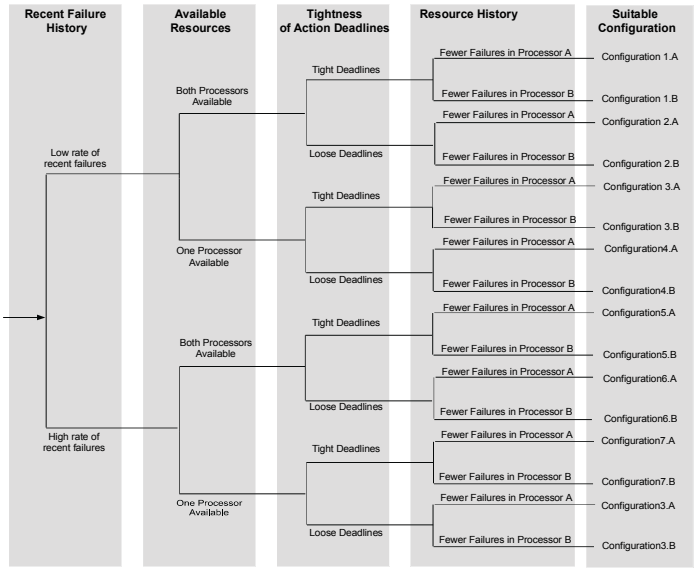

Figure 3-2. States for Adaptation Initiation Conditions

These modes were selected so that (i) the implementation of the AFT scheme should not incur an unacceptable additional cost and (ii) switching among various FT modes should increase the responsiveness of the system.

	Leader String			Follower String		
Config.	Speed	Execution Mode for Critical Tasks	Execution Mode for Non-critical Tasks	Speed	Execution Mode for Critical Tasks	Execution Mode for Non-critical Tasks
Mode 1	Single	Primary	----------	Single	----------	Primary
Mode 2	Double	Primary	----------	Single	----------	Primary
Mode 3	Single	Primary	Primary	-------	----------	----------
Mode 4	Double	Primary	Primary	-------	----------	----------
Mode 5	Single	Primary	----------	Double	Backup	Primary
Mode 6	Double	Primary	----------	Double	Backup	Primary
Mode 7	Double	Primary	Primary	-------	----------	----------
Mode 8	Double	Primary	----------	-------	----------	----------

Figure 3-3. Organizing Fault-Tolerant Configurations

Recovery can be classified into the following two categories:

(i) *Non-Replicated Execution Modes:* Non-replicated execution modes are the execution modes in which each task (Classes C or NC) is executed in, at most, one of the strings. In other words, none of the tasks is replicated in both strings.

(ii) *Replicated Execution Modes:* In Replicated execution modes, C-CLASS tasks are executed in parallel in both

strings A and B. This precautionary action is taken to avoid a long recovery latency.

Figure 3-4 depicts an overall structure of the AFT scheme and its embedded recovery mechanisms.

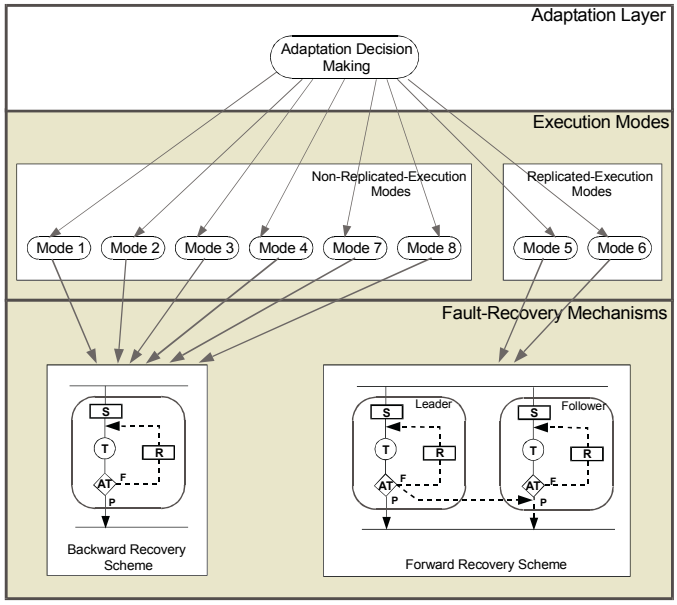

Figure 3-4. AFT Scheme and its Recovery Mechanisms

For recovery in non-replicated execution modes, we assume that transient faults can be detected by an acceptance-test component which is denoted in Figure 3-4 as *AT*. This can be realized in either hardware or software. The state of the processor is saved by the module S (the checkpointing module,) at the beginning of each cycle. If the *AT* accepts the result of the current cycle, then the output can be sent to its destination. However, if AT detects that a failure has occurred in the processor, then the the state of the system will be restored from the last checkpoint by R (the checkpoint restoration module) and the task will be reexecuted. In Figure 3-4, dashed lines depict the recovery actions. This technique is known as the Rollback and Restart or Backward Recovery mechanism. AT can be either a generic fault-detection module or an application-specific acceptance check. S and R are designed to save the current system state and to restore the last saved system state, respectively. Module S can be implemented as a generic function, such that it accepts (1) the addresses in which system state variables reside (this is application-dependent) and (2) the address of the location in which the state should be saved and to perform the checkpointing. The same technique can be applied for the module R.

For recovery in replicated modes, In this mode, the Class C applications are running in parallel in both leader and follower nodes. Both nodes perform checkpoints using S at the beginning of each cycle. If a transient failure occur in the system, one of the following recovery actions can be taken depending on the location of the failure:

- *Failures in the leader node:* If the leader node fails the *AT* test, then it notifies the peer node (the follower node) of its failure (by using a heartbeat message), and then rolls back and retries the execution of the Class C tasks. Moreover, it switches to become the new follower node. The reason for this switch is that behaviors of a recently failed node can no longer be trusted. On the other hand, the peer node receives file news about the failure of the leader node and immediately sends its already-accepted result to file destination and becomes the new leader node starting the next execution cycle. This technique is called the Forward Recovery Mechanism.

- *Failures in the follower node:* If the follower node experiences a transient failure, it rolls back and retries the execution of the task similar to non-replicated execution modes and notifies the leader node of its failures. The leader node updates the failure database on recent failure of the follower node but does not take any action on it**.** The reason for this passive recovery action is that the leader node is the producer of the result and the follower has enough time for a slow recovery action.

4. GENERIC MIDDLEWARE SERVICES

This section describes the Middleware services and higher-abstraction interfaces to the application components. Application designers can use these services without major concerns on how they are implemented. Figure 4-1 shows the conceptual view of the Middleware in the context of an application software layer.

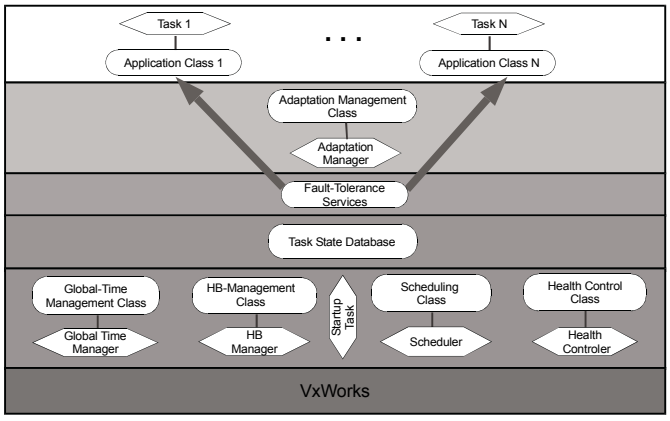

Figure 4-1. Middleware in the Context of the Overall Software System

The application layer can make use of specific fault tolerant services such as vehicle time or if they wish to terminate themselves (perhaps due to application detectable exception conditions such as missing deadlines or unanticipated data types). The application layer can also provide direction to the adaptation manager as described in the previous section in terms of the adaptation policy or identification of

resource constraints (e.g., malfunctioning sensor). The AFT Middleware in turn makes use of the operating system services, as well as mission specific services, such as power management or bus reconfiguration.

Table 4-1 lists the adaptation services and Figure 4-2 shows their relationship. The following sections describe each of these services in greater detail.

Table 4-1. Middleware services and their roles

`	**Purpose**	**Needs**
Vehicle time service	Maintain consistent time base.	Necessary for consistent behavior where timeout is primary detection mechanism.
Reliable messaging	Provide transparent method of replicated message transfer (and filtering).	Necessary for single-fault tolerant communications among software/hardware components.
Redundancy management	Status monitoring, active/shadow reconfiguration, and system restoration.	Automatic redundancy management and remote restoring previously failed resources are essential.
Replicated data management	Maintaining consistent state among replicated software components.	Essential for continuation of service upon failures.
Resource monitoring	Maintaining system status information	Supporting autonomy functions, and assisting in determining reconfiguration policies.
Data system management	Interface with vehicle autonomy and other high-level decision-makers and decide on the most suitable configuration for data systems.	Supporting subsystem level recovery.
Node state initalization and restoration	Restore the state of a node after a failure	Restoration of redundancy after a failure

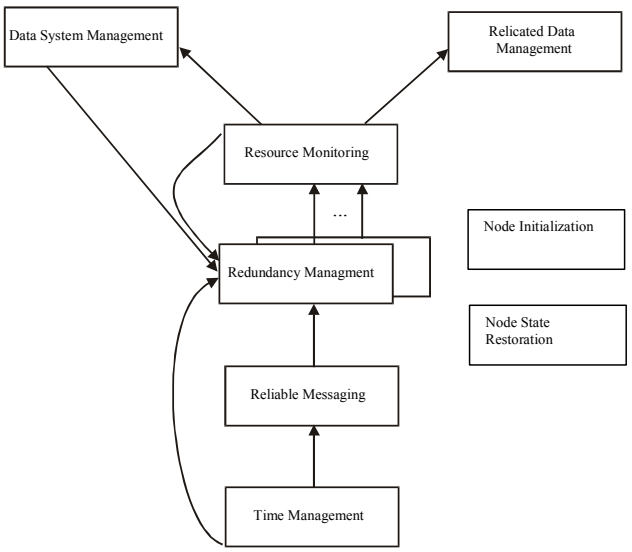

Figure 4-1. Relationship Among Services

Vehicle Time Service

The purpose of the vehicle time service is to maintain a consistent time base on applications that can perform real time calculations and in which the middleware can be used as a basis for timeout detection. It is necessary that all processes and processors have a uniform replicated clock on which such decisions can be made. In the test implementation, the hardware clock on the VME chassis was used as the reference time source by all processors. However, within a spacecraft, it is assumed that additional fault tolerance provisions will be added to the clock in order to prevent time from being a single source of failure.

Reliable Messaging

Commercial-off-the-shelf (COTS) operating systems (including VxWorks) perform intertask communications using low-level primitives, such as TCP or UDP socket-communication primitives. Socket-based communication is a low-level communication in the sense that the participant components must be aware of the implementation details, such as the locations of the components participating in the communication. Our objective was to facilitate a higher level abstraction for *reliable* communication among task (which may reside in different hosts) with the following characteristics:

- *Location transparent inter-task communication*: This implies that the sender task does not need to know the location of the receiver task(s) in order to communicate with them.

- *Logical channel based communication*: Messages can be sent to be received through a logical channel. Each task can join and leave a channel dynamically. When a message is sent to a logical channel, all registered parties will be able to receive the messages in the same order as which they were tranmitted into the channel.

- *Multicast communication*: The communication service provides one-to-many ordered communication among (remote) tasks.

- *Flexibility*: The communication subsystem provides both unreliable (best effort) and reliable channels. A message sent through a reliable channel will be delivered to the destination event in the presence of transient communication failures. This is done in an application-transparent manner.

Figure 4-3 illustrated the logical channel communication. In the rest of this subsection, we will discuss the main characteristics of the devised logical channel based communication and the relevant interfaces through which a component can communicate with others.

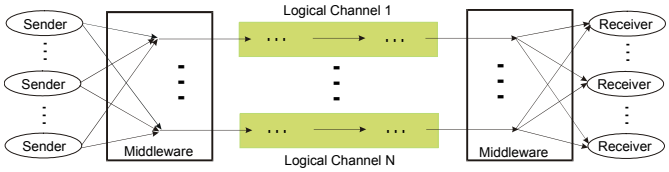

Figure 4-3. Logical channel based communication

A logical channel is a high-level abstraction to allow multi-sender multi-receiver communication among tasks residing on the same of different hosts. In our design, the middleware is responsible for the channel management (message transmission, message queuing, etc). If an application task is relocated to a different host, the operation of the logical channel will not be affected, since the middleware keeps track of the location of the node registered to each logical channel. There are two types of logical channel supported in the middleware:

- *Reliable logical channels*: When a message is transmitted through a reliable logical channel, the middleware guarantees that the message will be delivered to the alive destinations. The delivery mechanism is based on the concept of negative acknowledgement. In the case of a lost message, the middleware is the receiver side and will send a negative acknowledgement to the sender middleware, which will in turn retransmit the lost message(s).

- *Unreliable logical channels*: In this case, the message delivery is based on the best effort, and its delivery is not guaranteed.

The following important service calls are provided for logical channels:

- `ChannelRegister:` Processes register for messages by means of the logical channel
- `ChannelDeregister:` Processes stop message feeds from the logical channel
- `SendMessageToLogicalChannel,` Used by a process to send a message to all other entities subscribed to the logical channel
- `ReceiveMessageFromLogicalChannel,` Used by a process to accept a message to a message from another entity sent over the logical channel
- `CheckMessageOnLogiclChannel: Used` by a process to accept a message to a message from another entity sent over the logical channel

Redundancy Management

A fundamental issue in selecting candidate fault-tolerant execution modes, is to identify all credible types of anomalies in the selected application domains, hardware and software platforms. A minimal set of fault-tolerant execution modes should then be selected to facilitate sufficient tolerance capabilities for the identified fault types, while keeping the mode-transition manageable.

A major criterion for selecting the set of redunancy and fault-tolerance concepts to be supported by the middleware is their effectiveness for supporting required fault-detection and fault-recovery latencies. However, the resources required by each fault-tolerance scheme are also considered as a second selection criterion. The following fault-tolerance schemes are supported in the implemented middleware:

- Exception Handling
- Primary/Backup
- Sequential Recovery Block (SRB)
- Distributed Recovery Block (DRB)

Table 4-2 summarizes the attributes of these different modes which are discussed in further detail in the following subsections.

Table 4-2. Characteristics of Fault-Tolerant Execution Modes

Attribute	Exception Handler	Primary/Backup	SRB	DRB
Recovery Method	Forward recovery	Back-ward recovery	Back-ward Recovery	Forward Recovery (parallel execution of applica-tions)
Response Time	Short recovery latency	Relatively long recovery latency	Relatively long recovery latency	Short recovery, latency
Physical redundancy	No hardware redundancy	Explicit hardware redundancy may not be required	No hardware redundancy	Hardware redundancy
Multiple software versions	Application-specific exception handler is required.	Full version and a simple version of application are required	Dual versions may be required	Dual versions may be required

Exception-Handler (EH): Figure 4-4 shows the top level data flow for this implementation of software fault tolerance. Data is placed into a buffer called a recovery point, processed by a routine designated as "primary", and then output to an online runtime acceptance test (i.e., assertion check or post condition) which checks the output for acceptable results. Associated with each application task, such that in the case of any failure in the task, the associated exception handler is invoked. The result of the exception handler is the output of the task.

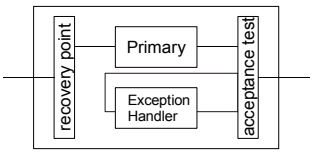

Figure 4-4. Exception Handler

Primary/Backup (PB): Under this scheme (Figure 4-5), each task is equipped with an acceptance test and exception handler. If an output of a method execution fails the acceptance test, then the associated exception handler is invoked. If the application task crashes or a hardware or operating system failure occurs, control is taken over by the follower node. The middleware is responsible for checking the heartbeat of all elements and ensuring that only one copy of the control task is in control. The middleware is also responsible for ensuring that any internally stored state data is kept consistent on both copies.

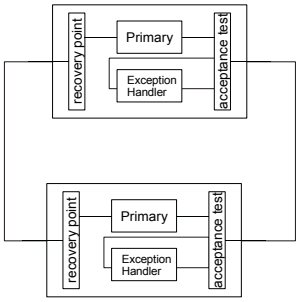

Figure 4-5. Primary/Backup Architecture

Sequential Recovery Block (SRB): A sequential recovery block [Rand75] consists of 3 major elements:

(1) A *primary routine*, which is analogous to the control routines, indicated as "primary" in the previous sections.
(2) An "alternate routine" which can perform the routine in a diverse and manner (e.g., use of a simple average as opposed to a root mean square)
(3) An acceptance test that is executed at each iteration during runtime and a primary and an alternate routine.

The primary routine takes the input data from the recovery point (i.e., a data buffer) and processes the data. The results are then checked by the acceptance test. If the results are acceptable, then the output is passed to the rest of the system. If not, the alternate routine takes the data from the recovery point and processes the data. The distinction between the alternate routine and the exception handler, described above, is that the alternate routine uses the same input data as the primary routine and repeats the processing step (albeit using an independent and/or lower risk algorithm). The exception handler performs a simpler operation without using the input data.

The basic control flow of the SRB is shown in Figure 4-6.

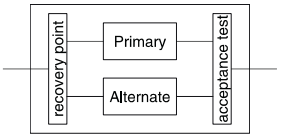

Figure 4-6. SRB Architecture

Distributed Recovery Block (DRB): The DRB [Kim95] is a replicated SRB as shown in Figure 4-7. The primary and alternate routine is executed in parallel. If the primary result fails the acceptance, the alternate result is immediately available on the parallel node. Therefore, the latency for any hardware and/or software failures affecting these tasks will be kept very short.

As was the case for the primary/backup scheme, the middleware is responsible for checking the heartbeat of all elements and ensuring that only one copy of the control task is in control. The middleware is also responsible for ensuring that any internally stored state data is kept consistent on both copies.

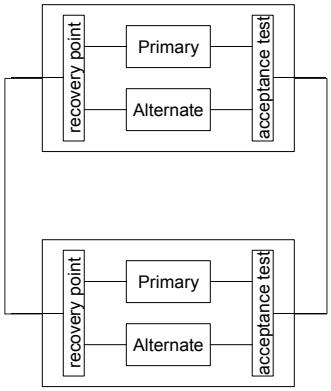

Figure 4-2. DRB Architecture

Replicated data management

The purpose of replicated data management is maintaining current consistent (*synchronized)* state data among replicated objects. It is necessary in order to maintain the continuity of services where state or history data is required (e.g., integrated data, configuration settings, etc.) There are several possible alternative methods to keep the follower processes synchronized with the primary, depending on the type of *state* data updates *received* by the receiving process and the *service* provided by the that process.

- Broadcast/multicast, i.e., All processes receive the same input data: This is facilitated through the logical channel service described above. All replicated processes register for the same incoming data, and all keep their state data updated in parallel. Additionally, period cross-check messages are circulated among the replicated processes to ensure that any broadcast or multicast messages have not been lost.

- Processing Input with Uniformity: Any state updates associated with incoming messages are made to all processes in the process before a response is generated. This technique can be used where allowed by response

time constraints.

- Processing Input Without Uniformity: this technique allows the process to generate a response without first ensuring that all processes in the process have updated their state data. In these cases it will be necessary for a recovering process to coordinate with a subset of its interfacing processes/objects in order to assure that the system state is consistent. This coordination process will involve a very small amount of data, since only the last set of state data updates will be suspect.

- Periodic Update: Each incoming State data update is sent by the primary process to replicas on a periodic basis. Again, a recovering process would have to coordinate with interfacing processes or objects.

The reliable messaging service discussed above, through its use of logical channel and multicast messages, provides the support for all of these methods.

Resource monitoring

The purpose of resource monitoring is to maintain the system status information needed by the middleware for reconfiguration and adaptation decisions. Such system status information is also needed by the telemetry function to provide status reports to the ground operations center, by the autonomy support functions for planning, and to make a higher level reconfiguration policy. In the middleware implementation, this was achieved using a resource failure history database. In the hierarchical recovery approach, described in the next chapter, the data may be encapsulated in subsystem or system-level fault handling objects.

Data System Management

The purpose of the data system management is to provide the interface with the vehicle autonomy and other high-level decision makers to provide input to the adaptation manager on the most suitable configuration for the data system, based on the mission phase and environmental conditions. It is also used by the telemetry function to provide status reports to the ground operations control center.

Node Initialization

The purposes of the node initialization service are:

- To control incremental activation of various services.
- Initiation of self-checking processes for components, such as buses.

The node startup service is highly system specific. For the Mission Data Sysem (MDS) on the The following initialization process was defined as a candidate for the Mission Data System (MDS) in the Pluto Kuiper Express::

1. Download a minimal kernel from the vehicle global NVRAM.

2. Perform initial diagnostics and self-checking (on components such as processor and memory). Find initial configuration of the system from local NV memory, non-local NV memory, or ground.
   - Available memory and configuration
   - Bus configuration
   - Initial mode of execution

3. Set system configuration parameters, such as:
   - Required speed for processor clock
   - Bus managers

4. Test all feasible bus configurations and determine which are available

5. Individual software service startup
   - Activate associated tasks
   - Inititalize states of the activated tasks
     Selective state initialization
     full stack initialization

Node State Restoration

The purposes of the node state restoration services are:

- To restart the failed or shutdown node.

- To determine the system configuration (e.g., replicated/non-replicated, high/low clock speed)and processor role (leader or follower).

- To provide additional startup configuration such as the (1) OS state (such as the open file descriptors), (2) States of system services, and (3) the state of each individual application.

The services are necessary to prepare the newly joined node as an up-to-date backup and if necessary for a node to be restarted due to a transient failure. Restoration can be done as a single event or incrementally. If done as a single event, the existing leader node would enter a minimum state so that it runs only essential functions and then provide the support to update the follower node. The update would consist of the state of system services and the state of each individual application. If backup is performed incrementally, the leader system can run in its normal mode. It provides state data information to the follower node incrementally, in order of criticality. That is, critical segments updated are passed initially, and then the state of the rest of the system will be transmitted as a lower priority task.

5. IMPLEMENTATION

This section describes the current implementation of the AFT middleware. The first subsection describes the basic architecture, the second identifies the major components,

and the third the integration with the application. Appendix A contains more details on the integration of applications with the middleware.

Basic Architecture

Figure 5-1 shows the top-level architecture for a dual redundant system (the architecture can be extended to greater levels of redundancy). The system is loosely coupled, i.e., each processor executes on a separate clock. Synchronization is achieved at the data frame level. Both processors receive input and execute in parallel, however, only one processor is responsible for output (for a dual/dual system, this is extended to processor pairs). The processor outputting data is designated as the *leader*, the processor which does not output is designated as the *follower*. The leader processor is responsible for making necessary adaptation decisions and configuring the whole system. The follower processor continuously monitors the health-status of the leader processor. In the case of a failure in the leader processor, the follower processor takes over the leadership responsibilities and executes the adaptation mechanism.

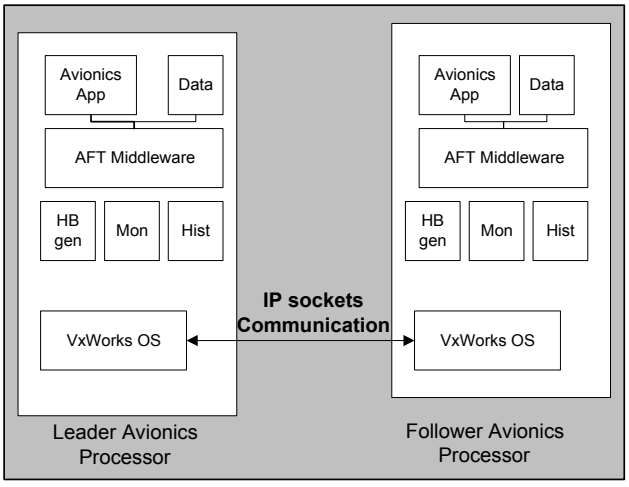

Figure 5-3. Top Level AFT Middleware Implementation

Within each processor, the operating system is responsible for executing a set of control tasks that may be incorporated into one or more operating system processes that execute at fixed iteration rates. Processes communicate by a combination of messaging and common memory. All critical state data are maintained in a designated global memory area referred to as the "data cache". At each iteration (or the finest grained iteration if multiple iteration rate processes are being executed), state data are read from the data cache and sensor data are read from either memory mapped I/O locations or from an avionics bus interface task. At the conclusion of the control process execution, state data are written back to the data cache and control outputs are written to the avionics bus management task or memory mapped I/O locations.

Components of the AFT System

The adaptive fault-tolerance system has the following components:

- *Heartbeat Generator:* Both leader and follower processors periodically send the heartbeat messages to the monitoring node, as well as to the peer processor.
- *Health-Control* Component: The Health-Control Component checks the health-status of its peer processor, as well as its own health, and reports the findings to Adaptation Manager.
- *Application* Components: This includes both critical and non-critical tasks.
- *Adaptation Manager:* The Adaptation Manager collects information from other components and decides on the most suitable fault-tolerance configuration for the whole system.
- *Resource History Database:* This database keeps track of failure and abnormal behaviors of system resources.
- *Environmental Database:* The Environmental Database keeps information on recent changes in the environmental conditions.

Integration with the Application

The frameworks and interfaces for integration of applications are available from the authors of this report. These frameworks include the capability to implement the application in any of the models described in using the Logical Channel and the software fault tolerance constructs are discussed in a separate document available from the authors.

6. TESTBED

This section describes the testbed for the AFT middleware. Section 7.1 describes the test system hardware and system software at the top level; Sections 7.2 and 7.3 describe the event injection and system monitoring components that interact with the middleware components described above.

Test System

The adaptive fault-tolerance demonstration was developed using the VxWorks. Figure 6-1 shows the hardware/software platform for the demonstration. Three MC68040 single board computers are located in a MVME167 chassis and can communicate with each other, as well as with the SPARC/ Solaris computer using the socket communication facility. In order to isolate fault domains of the processors, the shared memory provided by the MVME was not used.

Figure 6-2 depicts the overall architecture of the adaptive

fault-tolerant data system and its communication patterns with monitoring and simulation subsystems. The Monitor Subsystem provides GUI's for monitoring system status, as well as the application status. The Event-Injection subsystem provides facilities for (i) injecting various faults to different components, and (ii) activating conditions which lead to the occurrence of environmental changes.

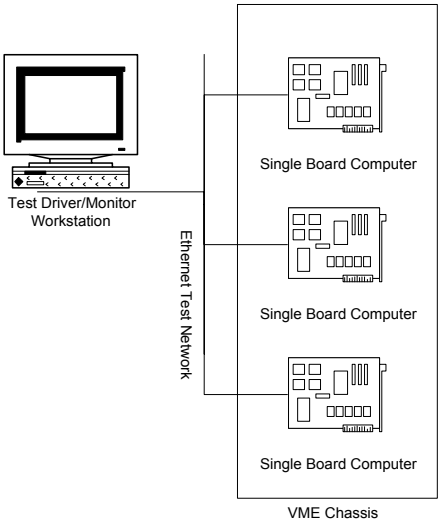

Figure 6-1. Hardware/Software Platform

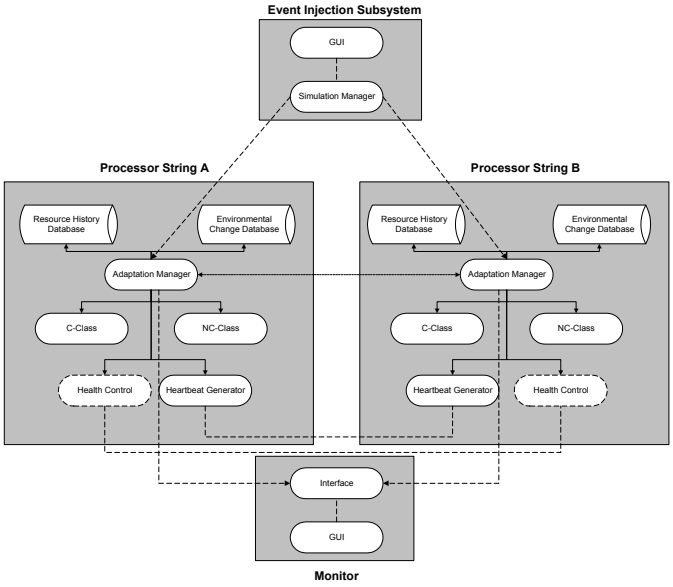

Figure 6-2. Testbed Software Architecture

Monitor Subsystem

The Monitor subsystem receives heartbeats and other status messages from both the leader and follower processors and displays the current status of the system, including the current configuration. Figure 6-3 presents a snapshot of the Monitor subsystem when the adaptive fault-tolerant system is in Configuration 6, in which the leader node is executing only the C-CLASS task (in single speed), while the follower node executes both C-CLASS and NC-CLASS task in the double-speed mode.

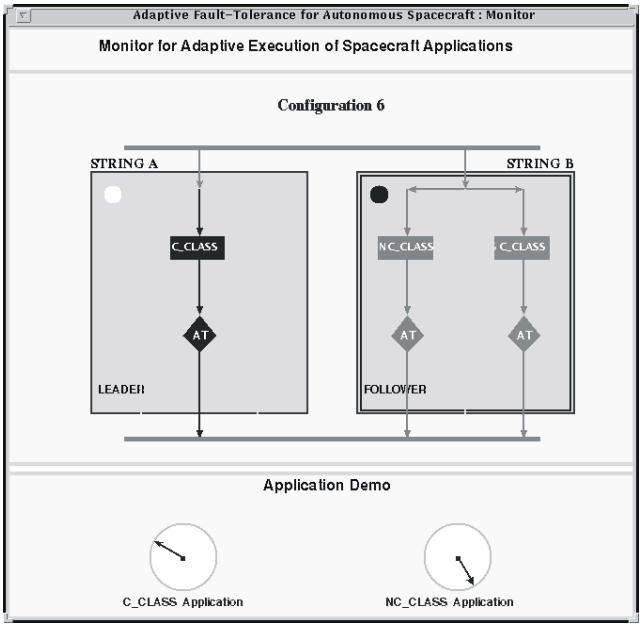

Figure 6-3 Monitor Subsystem Display

Event-Injection Subsystem

As shown in Figure 6-4, the Event-Injection subsystem is responsible for injecting events of two types into the systems: (i) occurrence of a failure in a component, and (ii) changes in the environmental conditions. For simplicity in the demonstration, environmental changes are denoted by increasing or decreasing the deadlines of C-CLASS tasks.

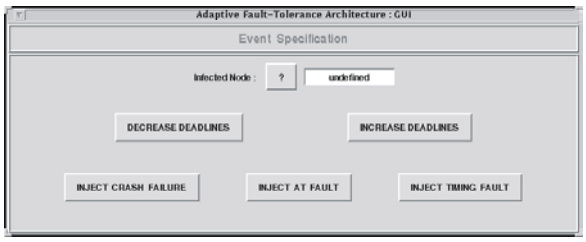

Figure 6-4. Event Injection Subsystem Display

CONCLUSIONS

Autonomous spacecraft systems have limited resources in power consumption, weight, and size constraints. However, varying environmental conditions, random events, and other factors necessitate a flexible and adjustable resource allocation and fault-tolerance strategy. The fault-tolerance provisions to be utilized in such systems must be able to adapt itself to changes in both (i) limit redundant resources and (ii) environmental demands and conditions. A fault-tolerance mechanism, which possesses such a desirable characteristic, is known as an adaptive fault-tolerance mechanism.

The research has demonstrated the feasibility of applying adaptive fault-tolerance concepts to spacecraft applications. We designed an adaptive fault-tolerance mechanism consisting of multiple fault tolerance strategies, which allow for the flexible utilization of hardware resources based on power constraints and its degree of hardware redundancy. In order to minimize resource utilization, while retaining specified reliability requirements of the system, the selected mechanism adapts to the changes in (i) available resources, and (ii) environmental conditions in order to minimize resource utilization while retaining specified reliability requirements of the system. Programming constructs and APIs were developed to facilitate the use of the middleware by spacecraft software developers.

We implemented a demonstration version of the designed adaptive mechanism in order to demonstrate the potentials of the adaptive fault-tolerance mechanism and to identify possible areas of concern. We also implemented an event-injection system which is responsible for (i) injecting various faults in the system and (ii) simulating changes in environmental conditions.

The result of the research demonstrates that adaptive fault-tolerance concepts can be applied to spacecraft applications with acceptable development costs and execution overhead.

ACKNOWLEDGEMENTS

This work was performed under the Small Business Innovative Research (SBIR) Phase I Contract NAS8-97037 from NASA. The authors wish to acknowledge the support and interest of Dr. Daniel Dvorak and Mr. Robert Barry from the X-2000 project.

BIBLIOGRAPHY

[Alk93] L. Alkalaj, "Advanced flight computing program', in *Proceedings of 1st Woprkshop on Advanced Fligth Computing and Industry*, Pasadena, CA, pp. 104-166, 1993

[Bro96] Brown, T., K., and Donaldson, J.A., "Fault Tolerance Architecture for the Cassini Spacecraft", *NASA Technical Brief, Vol. 20, No. 8 (also A JOL Technical Reprto).*

[Bar99] Barry, R., "Applying Architecture Concepts: Prototype S/C and Scenario", *JPL Internal Presentation*, Feb. 1999.

[Bro97] Brown, G.M., Bernard D.E., and Rasmussen, R.D., " Attitude and Articulation Control for the Cassini Spacecraft: A Fault Tolerance Overview*", A JPL Technical Report.*

[Dvo98a] Dvorak, D., "GAM Example Implementation", *Internal Presentation,* June 1998.

[Dvo98b] Dvorak, D., "Revisions for MDS Prototype", *Internal Presentation,* July 1998.

[Gat98] Gat, E., and Pell, B., "Smart Execution for Autonomous spacecraft*", IEEE Intelligent Systems*, Oct. 1998. Pp.56-61.

[Hec88] Hecht, M., and Fiorentino, E., "Causes and Effects of Spacecraft Failures", *Quality and Reliability Engineering International,* Vol. 4, No. 1, Jan. 1988, pp. 11-19.

[Hec93] Hecht, H. "Rare Conditions: An Important Cause of Failures", Proc. 1993 Computer Assurance and Safety Conference (COMPASS), Gaithersburg, MD, June, 1993

[Kim94] K. H. Kim, "Action-level fault tolerance," in *Advances in Real-Time Systems* (S. H. Sang, ed.), pp. 415-434, Prentice Hall, 1994.

[Kim95] Kim, K. H., "*The Distributed Recovery Block Scheme*", Ch. 8 in *Software Fault Tolerance*, R. Lyu Ed., John Wiley & Sons, 1995, pp. 189-210.

[Law95] Lawrence, T.F., "Anomaly management in complex systems", *Proceedings of 1995 Pacific Rim International Symposium on Fault Tolerant Systems,* Newport Beach, CA pp. 132-134, 1995

[Pel97] Pell, B., et al, "An Autonomous Spacecraft Agent Prototype", *NASA Ames Research Center Technical Report*, 1997.

[Ran75] Randell, B., "System Structure for Software Fault Tolerance", IEEE Transactions of Software Engineering, Vol. SE-1, No. 5, June 1975, pp. 220-232.

[Sho97] Shokri, E., et al., "Adaptive Fault-Tolerance for Military Planning Applications", A report prepared by SoHaR Inc. for USAF Rome Laboratory 1998.

[Sho98] Shokri, E., et al., "An Approach for Adaptive Fault-Tolerance for Object-Oriented Open Distributed Systems)", To appear in the *International Journal of Knowledge Engineering and Software Engineering*, 1998.

[Sta94] G. E. Stark, "Technology for improving the dependability of software-intensive systems: Review of NASA experience," in *Proc. Annual Reliability and Maintainability Symposium,* (Anaheim, CA), pp. 327-333, Jan. 1994.