FAULT-TOLERANT ARCHITECTURES FOR SPACE AND AVIONICS APPLICATIONS

Daniel P. Siewiorek and **Priya Narasimhan**
Electrical and Computer Engineering Department
Carnegie Mellon University, Pittsburgh PA 15213
dps@cs.cmu.edu, priya@cs.cmu.edu

Abstract

Over the past half century, computing systems have experienced over three orders of magnitude improvement in average time to failure and over seven orders of magnitude improvement in work accomplished between outages. This paper surveys, compares and contrasts the architectural techniques used to improve system reliability in space and avionics applications. The generic techniques are instantiated by actual system examples taken from the space and avionics domains. The paper concludes by observing trends and projecting future developments.

1. Introduction

Fault-tolerant space and avionics architectures have existed for the past 50 years, and have had considerable success in accomplishing their mission goals through rigorous architectural design, software engineering process, reliable implementation and testing. There tend to be several parallels between the two domains – for instance, their architectures tend to exhibit common themes in terms of error detection, redundancy for fault-protection, etc. However, the nature of the missions also differs across the two domains, and this influences their respective architectures to a great extent. Commercial aircraft are mass-produced and log many hours of flying on a daily basis with lifetimes of 20-30 years per aircraft – aircraft mission times are in the order of hours, with several human passengers on board the aircraft during each mission. On the other hand, spacecraft are typically "single-use" implementations and deployments, with mission durations in the order of months/years along with the need to sustain operation under harsh conditions and extreme thermal environments – furthermore, a large number of spacecraft missions (particularly those described in this paper) have been autonomous and unmanned.

In this paper, we provide a retrospective on the progression of some of the fault-tolerant architectures in the space and avionics domains, with the aim of understanding the concepts and motivation behind the current designs. In addition, we observe key trends (such as the use of commercial off-the-shelf products) in these domains, as well as discussing where the next generation of fault-tolerant space and avionic architectures should be heading.

2. System Failure Response Stages

A redundant system may go through as many as eight stages in response to the occurrence of a failure. Designing a reliable system involves the selection of a coordinated failure response that combines several reliability techniques. The ordering of these stages corresponds roughly to the normal chronology of a fault occurrence. It is important for the system designer to provide a response for each stage since the system will do something at each stage. It is better to have the system respond in a planned rather than an unplanned manner.

The following stages are typical in the failure response of any dependable system:

- **Fault confinement (containment).** This stage limits the spread of fault effects to one area of the system, thereby preventing contamination of other areas. Fault confinement can be achieved through the liberal use of fault-detection circuits, consistency checks before performing a function ("mutual suspicion"), and multiple requests/confirmations before executing a function. These techniques may be applied in both hardware and software.
- **Fault detection.** This stage recognizes that something unexpected has occurred in the system. Many techniques are available to detect faults, but an arbitrary period of time, called fault latency, may pass before detection occurs. Fault-detection techniques are divided into two major classes: off-line detection and on-line detection. With off-line detection, the device is not able to perform useful work while under test. Thus off-line detection assures integrity before and possibly at intervals during operation, but not during the entire time of operation. Off-line techniques include diagnostic programs. On-line detection provides a real-time detection capability that is performed concurrently with useful work. On-line techniques include parity and duplication.
- **Diagnosis.** This stage is necessary if the fault detection technique does not provide information about the failure location and/or properties.
- **Reconfiguration.** This stage occurs when a fault is detected and a permanent failure is located. The system might be able to reconfigure its components either to replace the failed component or to isolate it from the rest of the system. The component may be replaced by backup spares. Alternatively, the component may be switched off and the system capability reduced in a process called graceful degradation.
- **Recovery.** This stage utilizes techniques to eliminate the effects of faults. Two basic approaches are fault masking and retry. Fault-masking techniques hide the effects of failures by allowing redundant information to outweigh the incorrect information. In retry, a second attempt at an operation is made and is often successful because many faults are transient. One form of recovery, called rollback, backs up the system to some "safe" point in its processing prior to fault detection, so that operation recommences from that point. Fault latency becomes an important issue because the rollback must go far enough back to avoid the effects of undetected errors that occurred before the detected one. Issues in performing rollback-based recovery include dealing with irreversible operations, i.e., ones that cannot be undone (because they have consequences or side-effects that cannot simply be ignored or discarded) in the process of reverting to the "safe" rollback point.
- **Restart.** This stage occurs after the recovery of undamaged information. A "hot" restart, which is a resumption of all operations from the point of fault detection, is possible only if no damage has occurred. A "warm" restart implies that only some of the processes can be resumed without loss. A "cold" restart corresponds to a complete reload of the system, with no processes surviving.
- **Repair.** In this stage, a component diagnosed as having failed is replaced. As with detection, repair can be either on-line or off-line. In off-line repair, either the system will continue if the failed component is not necessary for operation, or the system must be brought down to perform the repair. In on-line repair, the components may be replaced immediately by a backup spare in a procedure equivalent to reconfiguration, or operation may continue without the component, as in the case of failure-masking redundancy or graceful degradation. In either case of on-line repair, the failed component may be physically replaced or repaired without interrupting system operation.
- **Reintegration.** In this stage, the repaired module must be reintegrated into the system. For on-line repair, reintegration must be accomplished without interrupting system operation. In some cases, upon reintegration, the repaired module must be re-initialized correctly to reflect the state of the rest of the functioning modules that it must now work with.

Table 1: Comparison of the commercial, space and avionics domains.

Operational Environment	Commercial	Space	Avionics
Mission duration	Years	Years	Hours
Maintenance Intervention	Manual	Remote	After mission
Outage response time	Hours	Days (Cruise phase)	Milliseconds
Resources - Power - Spare parts	 Unlimited Unlimited	 Minimal None	 Medium After mission
Fault-Tolerant Approach			
Fault avoidance and fault intolerance	Burn-in	Radiation-hardened components	Shake, rattle, roll
		Design diversity	Design diversity
		Safe system	
Fault tolerance		Component-level redundancy	
		Subsystem-level redundancy	Subsystem-level redundancy
	Multi-computer	Multi-computer	Multi-computer
	Retry	Retry	
	Firewalls		Firewalls
	Software patches	Software reload	

3. System-Level Approaches to Reliability

While the eight stages of fault handling are generic, the actual techniques and approaches to system reliability are a reflection of both the system's and the domain's operational environment. Table 1 compares and contrasts the operational environment for commercial, space, and aircraft systems. While the useful life of a commercial system is measured in years, in practice an aircraft computer system must only survive for the duration of a flight, at which point it could be totally replaced. Spacecraft computers must survive the entire duration of their mission since there is no opportunity for physical repair.

Even though commercial systems are moving more and more towards autonomous behavior, manual intervention is always a viable option for troubleshooting and repair. Physical intervention is available before and after a mission in aircraft systems. Unmanned spacecraft can be monitored remotely and the operations team can develop workarounds for problems, often on a ground-based replica of the flight system. These workarounds may take days to generate but time is available except in windows of planetary encounter. Commercial systems often undergo outages of an hour or more but aircraft systems must be able to handle problems in milliseconds, especially during critical take-off and landing phases.

For commercial systems, the availability of ground-based resources in terms of both power and spare components is practically unlimited. In spacecraft, power is at a premium and all the components have to be present during the initial launch. Aircraft systems can draw power from the engines but spare parts are often stored in repair depots that may only be visited after several flight segments based on the aircraft's routing schedule.

The approaches to fault tolerance have been specialized to suit these somewhat diverse operational environments that are specific to the domains. Larger commercial systems are subject to "burn in", an initial period of powered operation when temperature, voltage, and even clock frequency are varied while executing special diagnostic programs. The burn-in period for large systems may last

several days whereas for smaller systems, the first time power is turned on is when software is being loaded. Avionics and space systems add vibration testing to the burn-in phase. In addition, space systems use specialized components to withstand transient electrical fluxes that are experienced in space although, with higher flying aircraft, design requirements are also moving towards tolerating neutron-induced single-effect upsets. Both avionics and space systems tend to use design diversity, i.e., components and software of different designs to tolerate design failures. In addition, space systems are designed to enter a "safe" state and not attempt any irreversible actions such as the firing of rockets when there is any question of system integrity. This allows the ground team to develop strategies to be remotely uploaded, in the form of software, to the spacecraft to correct the initial situation.

In order to tolerate faults, spacecraft systems exploit redundancy at the subsystem level, and in some cases, even at the component level. All three environments use multiple computers that can share the load and offer the potential for graceful degradation. Retries are very effective in the transient-prone environment of space as well as in commercial systems. Both commercial and avionics system employ firewalls to avoid the propagation of harmful behavior. Software patches for ground-based commercial systems, and potentially entire software reloads for space systems, provide a costs-effective mechanism to recover from unanticipated situations.

4. Fault-Tolerant Software Architectures for Space Missions

4.1 Introduction

Spacecraft are the primary example of systems requiring long periods of unattended operation. Unlike most other applications, spacecraft must control their environment (such as electrical power, temperature, and stability) directly. Thus, one must consider all aspects of a spacecraft (for example, structural, propulsion, power, analog, and digital) when designing for reliability.

Spacecraft missions range from simple (such as weather satellites in low earth-orbit) to sophisticated (such as deep-space planetary probes through uncharted environments). Within this range are the following kinds of spacecraft: low earth-orbit communication or navigation, low earth-orbit scientific, synchronous orbit communication, and deep-space scientific satellites.

Each spacecraft is unique and specifically designed for its mission. Frequently, only one or two copies of the spacecraft are built. As an aid to understanding the specialized approaches to using fault tolerance in spacecraft, a generic spacecraft is described below, followed by detailed case studies of five specific spacecraft missions, again from a fault tolerance viewpoint.

4.2 Generic Spacecraft

A typical spacecraft can generally be divided into the following five subsystems.

- **Propulsion:** The propulsion system controls the stability and the orientation of the spacecraft. Multiple, often redundant, chemical or pressurized-gas thrusters are most frequently used. Occasionally, spacecraft employ a spin for stability instead of the active control provided by thrusters.
- **Power:** The generation and storage of electrical energy must be closely monitored and controlled because all other spacecraft systems operate on electricity. Most often, spacecraft electrical systems consist of solar cells and battery storage. The batteries carry the system through loss of sun or loss of orientation periods. Control of solar cell orientation, battery charging, power transients, and temperature is the most time-consuming task for the spacecraft computers. Nowadays, the handling of battery charging, power transients and temperature are usually handled by micro-controllers in the power subsystem.

- **Data Communications**: Data communications are divided into three, often physically distinct, channels. The first is commands from the ground to the spacecraft via the uplink. It is even possible to reprogram a spacecraft computer by means of the uplink. The other two channels are downlinks from the spacecraft to the ground. One downlink carries data from the satellite payload; the second carries telemetry data about the spacecraft subsystems (temperature, power supply state, and thruster events).
- **Attitude Control:** A dedicated computer is often used to sense and control the orientation and the stability of the spacecraft**.**
- **Command/Control/Payload**: All aspects of spacecraft control are usually centered in a single command/control computer. This computer is also the focus for recovery from error events. Recovery may be automatic or controlled from the ground via uplink commands.

Typically, each subsystem is composed of a string of stages. As an example, a representative power subsystem consists of seven stages. Solar panels are physically oriented by tracking motors. Power is delivered to the spacecraft via slip rings. A charge controller automatically keeps the batteries at full potential, a power regulator smoothes out voltage fluctuations; and a power distributor controls the load connected to the power subsystems. The granularity of the power distributor's ability to switch loads is closely tied to the space system's fault containment and reconfiguration capability. The smaller the load that the distributor is able to switch, the finer is the containment and reconfigurable regions, but the cost is also higher. At each stage, redundancy is used to tolerate anticipated fault modes. To reduce complexity usually only the output of a string is reported via telemetry.

NASA's Preferred Reliability and Maintainability Practices document [17] defines *fault protection* as "the use of cooperative design of flight and ground elements (including hardware, software, procedures, etc.) to detect and respond to perceived spacecraft faults." This document details proven fault-tolerance strategies and architectures for generic spacecraft – these were certified for multiple missions, specifically, for Voyager, Magellan, Galileo and Cassini. As this document describes, typical spacecraft architectures focus on avoiding single points of failure and on preserving system integrity in the face of anomalous events.

In the remainder of this section, we will consider the fault-tolerant designs for five specific space missions: (i) RCA's Defense Meteorological Satellite Program (DMSP), which relays weather photographs from a polar orbit, (ii) Voyager, which is a deep-space probe used in the Jupiter and Saturn planetary fly-bys, (iii) Galileo, a Jupiter orbiter and probe mission, (iv) Cassini-Huygens, a Saturn orbiter and probe mission, and (v) Pathfinder, a Mars planetary lander carrying an autonomous mobile rover.

4.3 Defense Meteorological Satellite Program (DMSP)

We will use DMSP, a simple spacecraft, [23] as a running example to illustrate the discussion of generic spacecraft architectures, redundancy techniques, and error-management procedures. Each spacecraft has a unique architecture for interconnecting the generic subsystems as well as for interfacing the subsystems to the payload designed to carry out the mission of the spacecraft. Figure 1 depicts the interconnection of the generic subsystems in DSMP with the telemetry and meteorology sensor subsystems. (The propulsion subsystem has not been shown in order to simplify the example.)

A standard set of redundancy techniques, each tailored to a generic subsystem, has evolved through several generations of spacecraft. A representative set of techniques for each generic subsystem includes the following:

- *Propulsion:* Redundant thrusters, including multiple valves for propellant flow control, automatic switchover based on excessive attitude-change rates, and multiple commands required to initiate any firing sequence
- *Power:* Redundant solar cell strings, batteries, power buses; automatic load shedding

- *Data Communication:* Redundant transponders, digital error-detection and correction techniques, switch from directional to omnidirectional antennae for backup
- *Attitude Control:* Redundant sensors, gyros, and momentum wheels, along with automatic star reacquisition modes
- *Command/Control*: Redundant computers, memories, and I/O interfaces; hardware testing of parity, illegal instruction, memory addresses; sanity check; memory checksums; task completion timed; watch-dog timers; memory write protection; reassemble and reload memory to map around memory failures.

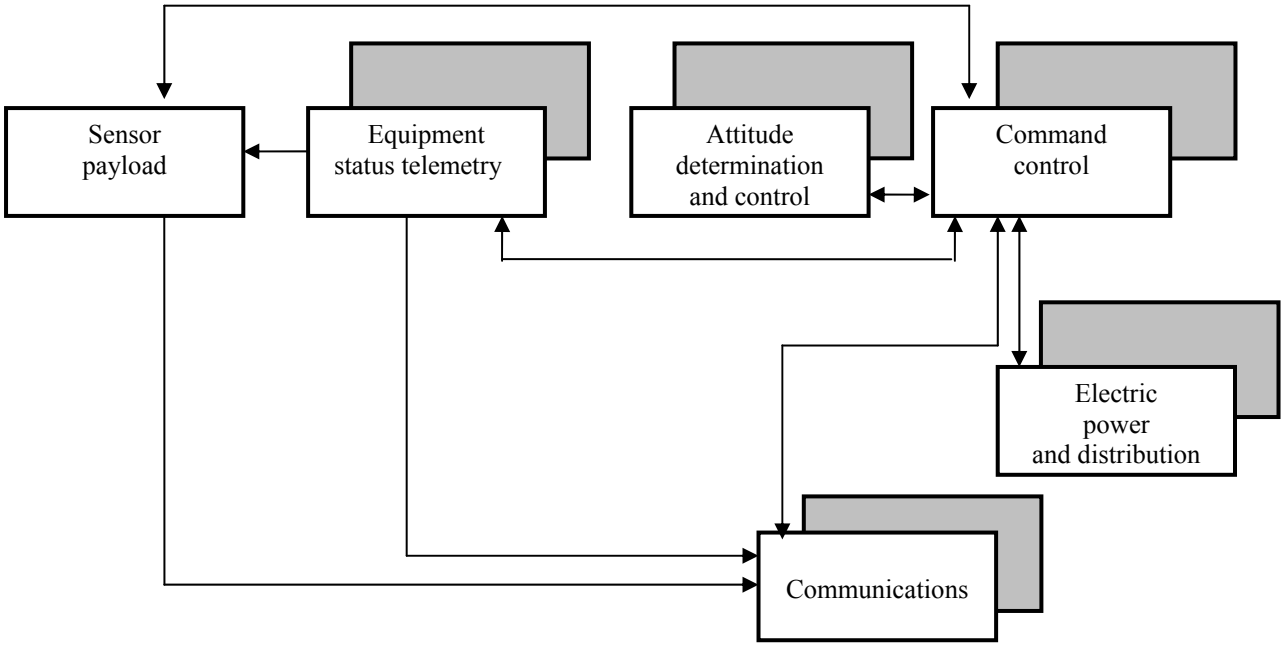

Figure 1: Interconnection of major subsystems in RCA's Defense Meteorological Satellite Program (DMSP) block 5D-1 spacecraft.

Returning to our specific example of DMSP (Figure 1), we see that standby redundancy is used in all but the sensor payload. The standby spares are cross-strapped so that either unit can be switched in to communicate with other units. This form of standby redundancy is called *block redundancy* because redundancy is provided at the subsystem level rather than internally to each subsystem.

When an error is detected, most spacecraft systems enter a unique error-management procedure called a "safe" or "hold" mode. As a part of the spacecraft-safing process, all nonessential loads on the power subsystems are shed; in addition, normal mission sequencing and solar array tracking are stopped. The spacecraft's solar panels are oriented to obtain maximum solar power while the spacecraft awaits further command sequences from the ground. Meanwhile, ground personnel must infer which failures could cause the output behavior of each of the strings, and then devise a possible failure of the spacecraft subsystems. A command sequence implementing the work-around is then sent to the spacecraft. Depending on the failure's severity, this procedure may take days, or even weeks, to complete.

Response to faults in generic spacecraft varies from automatic in hardware for critical faults (such as those related to power, clocks, and computers), to on-board software for serious faults (such as those related to the attitude and command subsystems), to ground intervention for non-critical faults. Faults can be detected by one of several means:

- *Self-Tests:* Subsystems perform self-tests, such as checksums on computer memories.
- *Cross-Checking Between Units:* Either physical or functional redundancy may be used. When a unit is physically duplicated, one is designated as an on-line unit and the other as a monitor. The monitor checks all the outputs of the on-line unit. Alternatively, there may be disjoint units capable of performing the same function. For example, there is usually a set of sensors and actuators for precision attitude control. Attitude may also be less precisely sensed by instruments with other primary functions. The less precise calculation can be used as a sanity check on the more precise units.
- *Ground-Initiated Special Tests*: These tests are used to diagnose and isolate failures.
- *Ground-Trend Analysis*: Routine processing and analysis or telemetry detect long-term trends in units that degrade or wear out.

DMSP uses block redundancy, cross-checking on attitude control, routine self-testing, automatic load shedding upon under voltage detection, and block switching under ground control. Internally detected error conditions include memory parity, memory address, arithmetic overflow, and illegal transfer.

With this background, we can examine the evolution of multiple generations of spacecraft and planetary probes.

4.4 Voyager

The two Voyager spacecraft missions launched in 1977 were designed to investigate four of the five outer planets of the solar system. It was designed as a "fly-by" spacecraft. Typically, the planetary encounter phase of each mission lasted 100 days, and was composed of a 30-day "observatory" phase (regular periodic observations of the planetary system), a 30-day "far-encounter" phase, (observation of the planet's satellites and spacecraft reorientation in order to calibrate various field and particle sensing instruments), a 10-day "near-encounter" phase (high-resolution observations, intense data gathering, and Sun/Earth occultation), and a 30-day "post-encounter" phase similar to the initial "far-encounter" phase. Between encounters, the spacecraft calibrated instruments, gathered interplanetary "cruise science" data, such as field and particles information, and prepared for the next encounter.

The Voyager spacecraft primarily used block redundancy for fault tolerance [10]. The attitude control subsystem (ACS) is composed of redundant computers, with one being an unpowered standby spare. The command and control subsystem (CCS) is also a redundant computer, but the standby is powered and monitors the on-line unit. Cross-strapping and switching allow reconfiguration around failed components. The CCS executes self-testing routines prior to issuing commands to other subsystems. New programs for memory must be loaded from the ground.

Error detection in the Voyager ACS is composed of the following:

- Failure of CCS to receive "I'm-Healthy" report every 2 seconds
- Loss of celestial (Sun and Canopus) reference
- Failure of power supply
- Failure to rewrite memory every 10 hours
- Thruster failure (spacecraft takes longer to turn than expected)
- Gyro failure
- Parity error on commands from CCS
- Incorrect command sequence
- Failure to respond to command from CCS

Error detection in the CCS includes the following hardware and software tests:

- Hardware: Low-voltage; primary command received before previous one processed; attempt to write into protected memory without override; processor sequencer reached an illegal state
- Software: Primary output unit unavailable for more than 14 seconds, self-test routine not successfully completed; output buffer overflow.
- Special Reed-Solomon coding.

4.5 Galileo

A follow-on to the Voyager Jupiter fly-by mission was the Galileo Jupiter orbiting and probe insertion mission. The Galileo architecture [11] borrows heavily from the experiences gained with the Voyager system. Block redundancy is used throughout the ten subsystems comprising the orbiter. All but the command and data subsystem (CDS) operate as an active/standby pair. The CDS operates with active redundancy, wherein each block can issue independent commands or the blocks can operate in parallel on the same critical activity. The major departure from the Voyager architecture is the extensive use of microprocessors in the Galileo orbiter. A total of nineteen microprocessors with 320 KB of random-access memory and 41 KB of read-only memory form a distributed system communicating over an 806 kHz data bus. Nine scientific instruments add eight further microprocessors to the total system. The bus is used to pass network-like messages between sources and destinations. Due to the volatile nature of the RAM, a further requirement for the Galileo orbiter has memory Keep-Alive inverters to maintain power to the CDS and to the attitude and articulation control subsystem (AACS) memories in case of power faults. The orbiter accommodates a total of nine scientific instruments (five for fields and particles and four for remote sensing science). Due to the nature of the phenomena to be measured, the field-and-particle instruments demand a spinning platform to make total spherical observations, while the remote-science instruments require very accurate and very stable pointing. These requirements produced a dual-spin structure in which a portion of the spacecraft is spun at three to ten revolutions per minute, while the other portion is held in a stable fixed configuration.

Power is transmitted across the spun/despun interface via slip rings, while rotary transformers are used for data signals. The downlink data rates vary as a function of the experiment. Nonimaging science experiments require a high-quality bit-error rate (less than five times 10^{-5}) and are encoded using a Golay (24, 12) error-correcting code. Imaging experiments can use a lower quality bit-error rate (less than five times 10^{-3}). The orbiter was designed to operate reliably and autonomously in the harsh Jovian radiation and electrostatic-discharge environments during the critical phases of relaying probe data and orbit insertion.

The Galileo spacecraft has few hardware error-detection mechanisms. Faults are detected by monitoring the performance of various spacecraft subsystems. The following is a partial list of error-detection mechanisms [8]:

- Test of event durations including transfers between subsystems and transition between all spin and spun/despun modes
- Parity or checksum errors on messages
- Unexpected command codes
- Loss of “Heartbeat” between the AACS and the CDS
- Spin rates above or below set values
- Loss of sun or star identification detected by no valid pulse from acquisition sensor for a given period of time
- Too great an error between control variable setting and measured response.

The on-board fault-protection software is designed to alleviate the effects and symptoms of faults rather than to pinpoint the exact sources of faults themselves. Fault identification and isolation is performed by ground intervention.

It's worth noting that Galileo's high-gain antenna failure almost made the mission unworthy. The recovery from that failure occurred through the use of a low-gain antenna combined with a data compression technique. Ultimately, about 70% of the science data was recovered.

4.6 Cassini-Huygens

Cassini-Huygens is a $3.26 billion joint NASA / European Space Agency (ESA) mission to study Saturn, its rings and moons. Launched in 1997, the Cassini spacecraft arrived at Saturn in July 2004. The Huygens Probe was released and landed on 14 January 2005 on Titan, Saturn largest moon, to make a wide variety of scientific measurements, while the Cassini Orbiter overflies it. The Cassini mission involved an interplanetary journey of 6.7 years, during which Huygens was to be dormant, except for health checks every six months, until it was activated for its 2.5 hour data-collection mission on its descent to Titan. The probe data-relay subsystem serves as the umbilical link between the probe and the orbiter during the probe's descent, while the Command and Data Management Units (CDMUs) manage the probe's mission particularly when it can no longer be tele-commanded. We discuss below the fault tolerance architectures of both the Huygens probe as well as the Cassini orbiter.

The Huygens probe [5] comprises various subsystems, including those to oversee the mechanical/thermal components, the electrical power, probe-relay data, along with command and data management. The probe has multiple mission modes – cruise, coast, entry, descent – that dictate the states of its various subsystems.

The overall philosophy behind the Huygens probe's fault-tolerant design is autonomy – since the probe cannot be commanded after separation from the orbiter. The electrical architecture is wired for complete redundancy. The fault-tolerant software architecture uses block redundancy, and exploits two identical Command and Data Management Units (CDMUs), a triply redundant Mission Timer Unit (MTU), two mechanical g-switches (backing up the MTU), a triply redundant Central Acceleration Sensor Unit (CASU) include dual-redundant accelerometers and proximity sensors. Each CDMU executes its own software simultaneously in a hot-backup configuration, where each replica can run the mission independently. The CDMU replicas are identical in almost all respects, with minor differences to avoid common-mode failures, e.g., one replica's telemetry is delayed by 6 seconds to avoid data-loss in case connectivity of the telemetry link temporarily suffers. Error detection is performed by both replicas of each of these units. Either replica considers itself invalid if it detects the following: a two-bit error in the same memory word, an Ada exception, or an under-voltage on the CDMU power-line. Redundancy is considered not only for online units, but also for the scientific data from the mission. To ensure that experimental data gathered is preserved, the data from the probe is also redundantly mirrored within the orbiter for later downlinking and transferred to Earth. Mission and profile reconfiguration are possible only through a write to specific data tables stored in EEPROM

The fault-protection mechanisms [19] in the Cassini orbiter architecture were designed with the typical reliability requirements of spacecraft, but with additional considerations due to the unusually long 11-year mission time. The overarching goal of the Cassini System Fault Protection (SFP) was to design the system to ensure that there were "no single points of failure" anywhere [19]. Additional goals involved system recovery and reconfiguration from multiple faults, provided that these faults occurred in independent fault-containment regions. In the interests of controlling software complexity and simplifying dependability, the Cassini SFP adopted a priority-driven one-fault-at-at-a-time fault-recovery approach. In the recovery and reconfiguration process, multiple faults could be assigned the same priority, with identical-priority faults being handled in a first-come-first-served basis. Typically, the highest priority was assigned to the most time-critical faults, while the lowest priority was assigned to those fault-recovery actions that took the longest to complete.

The responsibility for fault-recovery was divided between the spacecraft and ground operations; however, autonomous time-constrained fault-recovery was provided onboard the spacecraft because timely ground response could not always be expected. Each subsystem was designed to be self-recovering, with the intervention of the SFP where this was not possible. The Command and Data

Subsystem onboard the orbiter served as the host for the SFP, and was implemented in a passive replication (primary-backup) scheme. The backup CDS replica was powered off for the majority of operations, and was activated by the error-detection mechanisms in the watchdog timer in case the primary CDS replica failed to reset the timer every 32 seconds. An entire system of watchdogs was responsible for monitoring the viability of all of the subsystem processors in the spacecraft. The CDS of Cassini was designed to support a hot-backup configuration and both strings were powered. Each string performed a periodic self-check. If the result of the self-check was normal, it toggled a "healthbeat" signal to the other string. If the backup string detected the loss of healthbeat from the primary, it promoted itself to become the new primary.

One goal was that the inadvertent execution of a fault-recovery action should not be hazardous to the mission; thus, recovery was designed to be interruptible without causing harm to the spacecraft. This was facilitated by ensuring that the cancellation/interruption of a fault-recovery action triggered a "safing response" by entering the system into a low-power state, and configuring propulsion, heating, and telecommunication subsystems to maintain the spacecraft within safe operational limits.

The fault-recovery actions were performed in the order of increasing severity. The fault-recovery action depended on the mission mode (mission phase, in-flight history, etc.) and the environment (runtime performance of spacecraft hardware).

4.7 Mars Pathfinder

NASA's missions to Mars have evolved in successively more difficult stages: flyby, orbiter, lander, and rover. The very first longer term, global studies as missions flown by Mariner were picture-taking flybys. Orbital Missions have included the Mars Climate Orbiter, Mars Global Surveyor, and Mars Reconnaissance Orbiter. Lander missions have included the Viking 1 and Viking 2 while rover missions have included Mars Pathfinder and Mars Exploration Rovers. We will focus on the fault-protection and reliability techniques of the Mars Pathfinder – the landing of the Pathfinder on Mars set the stage for the release of Sojourner, a rover intended to explore the Martian surface.

Mars Pathfinder [15] was a NASA Discovery Mission launched in 1996 emphasizing the "faster, better, and cheaper" NASA vision [24]. The purpose was to demonstrate NASA's commitment to low-cost planetary exploration through small, successful, challenging missions built on a tight schedule and a budget cap. The Pathfinder mission stands in sharp contrast to the earlier Viking 1 and 2 landers – (i) Pathfinder cost a total of $280 million dollars including the launch vehicle and mission operations, while the development of the Vikings cost $3 billion 1997-dollars, (ii) Pathfinder was developed in four years compared to the eight years for Viking, and (iii) the Sojourner rover worked for a month, the Pathfinder lander for a year, while the Viking landers lasted six years.

From a technical perspective, the Pathfinder represented a series of significant firsts – it was the first spacecraft to fly straight into a planet's atmosphere and land on the planet without orbiting, the first to use an airbag (similar to automotive airbags) to cushion the impact of the landing, and the first to deploy a rover, a semi-autonomous instrument-carrying mobile robot tele-commanded by remote control from earth with a mission to chart the composition and size of Martian rocks, dust and debris.

The Mars Pathfinder employed extensive testing and block redundancy. The Mars Pathfinder combines the computers for Command and Data Handling and Attitude Control into a single computer called the Attitude and Information Management (AIM) computer. AIM is a single string design and there is no block redundancy. That was a conscientious decision due to the "faster, better, cheaper" philosophy. The fault protection is solely relied on software and a large number of fault monitors were implemented in software. The main fault recovery mechanism is processor reset and graceful degradation. Since software fault protection is slow, it is disabled during the time-critical entry, descent, and landing (EDL) phase. Risk mitigation was ensured through periodic system failure-mode and fault-tree analyses that were revisited/updated throughout the development process. The Mars Pathfinder flight software was written in C using object-oriented design principles. The code was stored in 2Mbytes of

EEPROM, with a backup EEPROM containing the entire software to allow fault-recovery in case of a burn-in or a software-upload gone awry. All of the flight software was developed by a core team of eight people.

The "common-sense" mission-assurance mechanisms [4] in the Mars Pathfinder were somewhat of a departure from previous NASA practices, and were dictated by key mission characteristics: short mission duration (7 months cruise, 1 month surface operations), budget cap, high entry and landing accelerations, and severe thermal extremes (-100 degrees Celsius at night and 20 degrees Celsius during the day) on Mars:

- *Short mission duration*: allowed for the higher risk of using Grade 2 (Class B) MIL 883B as the minimum parts quality for the spacecraft, as compared to the Grade 1 (Class S) parts used in Voyager. Because of the short mission, it was possible to use selective, rather than complete, block redundancy. While some subsystems are operated in a dual-redundant mode, other subsystems operated in "single-string" mode.
- *Budgetary constraints*: Cost savings were realized by eliminating incoming inspections on parts that cost less than $100 since studies had demonstrated a low return for such inspections. Effectively, only ~20% of incoming parts were inspected. Documentation was dramatically reduced, as were some expensive reliability analyses, such as Failure Mode Effect Criticality Analysis (FMECA) at the circuit level. But FMECA was retained at the interface subsystem level. The vendor's quality assurance methods were exploited as far as possible. The cost of the failure reporting (P/FR) subsystem was reduced by maintaining an electronic log of pre-mission problems, with specific, critical problems elevated to "formal" P/FRs only if the problem was assessed to have a significant adverse impact on the mission. Thus, Pathfinder recorded only ~200 P/FRs, compared with other missions (~1000 P/FRs being normal, ~3000 for Cassini, and 4000+ for Galileo). Closure of formal P/FRs was done on a regular and concurrent basis, typically involving a member of the mission assurance team, and proved to be less expensive ($3K each, vs. $10-20K for traditional practices). The system was so successful that it now forms the basis of JPL's institutional Problem and Failure Reporting System.
- *Severe landing environment and extremes of thermal cycling*: Most of the previous NASA missions tended to defer environmental testing to the system level; however, given the demanding thermal and entry/landing conditions for the Pathfinder (both of which were features that had not been previously encountered in any mission), assembly-level, subsystem-level and system-level testing were employed.

Concurrent engineering practices and the collocation of personnel were key ingredients in the quality assurance whose total budget for hardware and software was only $5M. While cost-cutting reliability practices were used, they were based on reasonable design choices derived either from prior experience and statistics of other missions without compromising system safety.

5. Fault-Tolerant Software Architectures for Commercial Avionics Systems

5.1 Introduction

Avionic systems share some of the same concerns as spacecraft systems. Avionic systems must maintain the environment of the aircraft within the structural limits designed for the airframe as well as flying limits (e.g., stalling) dictated by aerodynamic laws. While weight and power consumption are concerns as in spacecraft, they are not nearly as severe, and powered replicated components are used to increase safety and reliability. Whereas spacecraft are often "one of a kind" systems, several hundred copies of successful commercial aircraft designs are mass-produced.

Avionic systems have tracked trends in the commercial computing market by adopting similar concepts even though they may not incorporate the hardware and software commercial components derived from those concepts. The avionics designer must factor in the long aircraft life-cycles (typically 25 years or more) versus the much shorter commercial life-cycles. It is worth noting that aircraft avionics systems can be periodically maintained. This is a major consideration in fault-tolerance design of these systems since the system design can focus on extremely high reliability for a relatively short time.

5.2 Generic Avionic System

When electronics were first applied to aircraft flight control, there was a separate electronics box for each control function. As computers became more capable, functions were combined, thereby reducing the number of computers required. Physically dispersed redundancy has historically been the approach to tolerate physical damage and functional failure. Often two or more completely redundant paths are provided. For example, two generators driven by two different main aircraft jet engines would provide power to two independent computers that, in turn, would drive two different hydraulic systems for controlling the flight surfaces. In addition, functional redundancy provides additional alternative means for achieving each function. For example, if both jet engine generators were to fail, batteries would provide backup power while a ram air turbine would deploy automatically into the aircraft slipstream, thereby providing enough electricity to start an auxiliary generator.

Over the past two decades, avionics flight control has been moving from a partial "by-wire" (flight controls in both mechanical and electronics components) to a more complete "by-wire (flight controls fully in electronics) architecture. Figure 2 shows a high-level overview of an aircraft's electronic flight control system (EFCS) interactions within the aircraft. In this section, we discuss two representative civilian aircraft architectures: the Airbus A330/A340/A380 series and the Boeing 777. The discussion of military aircraft is outside the scope of this paper.

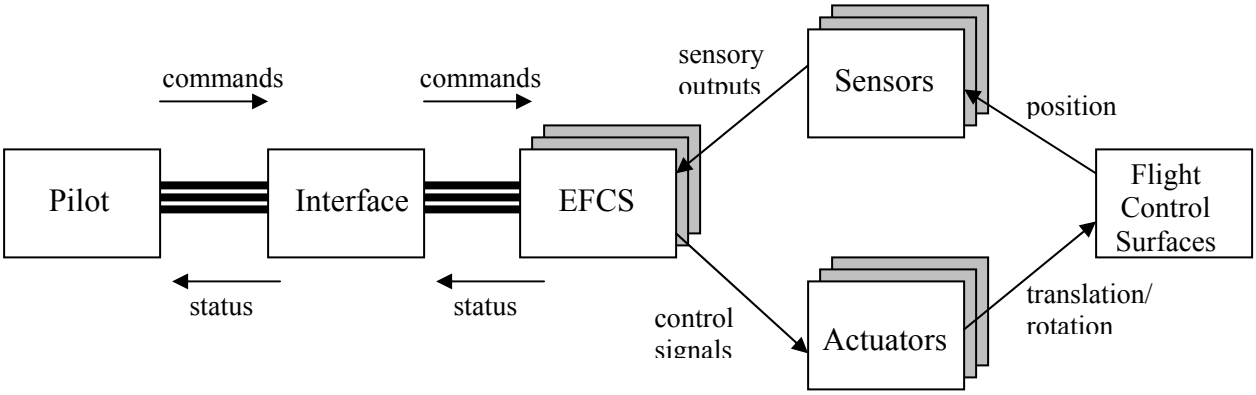

Figure 2: Generic avionics architecture, showing the electronic flight control system (EFCS).

5.3 Airbus A330/A340/A380

Introduced in 1983, the A310 was the first Airbus to have digital flight control, where ten separate computers were required. The Airbus A320 introduced fly-by-wire (e.g., signals from the cockpit to control surfaces were sent electronically rather than mechanically or hydraulically) in 1988. Four computers were teamed in command/monitor pairs that became the standard model for subsequent Airbus flight control computers. By 1992, the Airbus A340 had integrated all of the flight controls into one command/monitor computer pair. In the command/monitor pair, the command computer generates orders while the monitor computer is in more of a passive, observing role. The command/monitor

computers compare differences in the commands that they produce to a predetermined threshold. The differences between the commands must last for a sufficiently long period before the command/monitor pair disconnects forming a "fail fast" module. Another command/monitor pair is a standby "hot spare". In addition to mismatches, sequence checking (e.g., whether the tasks are sequenced in a predetermined order triggered by clock-generated interrupts) also signals the successful completion of actions. The computers also run a self-test whenever the aircraft is energized – this happens at least once a day. Another critical aspect of the reliability strategies is that alarms, fault-notification messages and fault-recovery actions are provided and executed in real-time in order to allow for timely and safe compensation to failures.

In addition to physical and environmentally induced errors, the Airbus computers also consider design and manufacturing errors. In addition to the two primary command/monitor computers, there are two secondary command/monitor computers based upon a different, simpler hardware microprocessor. Each of the four pairs has different software developed using different tools and different software development teams. The independence of the various components, distinct and physically segregated, reduces the risk of common-mode and common-area failures. *Design diversity* is, thus, an important aspect of these systems, involving the widespread use of dissimilar computers, physical separation of redundant entities, multiple software bases, different compilers for development, and data diversity.

The Airbus A380 continues the incorporation of concepts from commercial computing by using dual-redundant Ethernet data networks and PC/Windows for non-critical applications such as passenger entertainment, passenger list for flight attendants, and the flight log for the cockpit crew.

In the A330/A340, the flight control actuators were controlled by a hydraulic subsystem while the avionics were powered by the electrical subsystem. In the A380/A400, the flight controls and actuators span both the electrical and hydraulic generation subsystems, thereby providing more redundancy, increased segregation and dissimilar (hydraulic/electrical) power sources.

5.4 Boeing 777

The Boeing 777 has stringent dependability requirements [29] including some degree of tolerance to Byzantine-faults in order to deal with asymmetric faults (disagreement between replicas) in the functional and communication operations.

In addition to tolerating tradition faults such as object impact, failure of electrical components, failure of electrical power, interference by electromagnetic/lightning/radiation and cloud environment in the atmosphere, the designers of the Boeing 777 had a goal to increase the Mean Time Between Maintenance Actions to 25,000 operating hours. Another goal was to reduce the probability of degrading below minimum capability to less than 10^{-10}. As a result, the primary flight computer has three independent channels each composed of three redundant computing lanes. The triply redundant computing lanes consist of command, monitor, and standby computers. The standby computer allows for the dispatching of the aircraft even with one failure in a lane or with one of the three data channels failed. Arbitrary faults are dealt with through simple but effective Byzantine-fault tolerance (instead of the strong consensus protocols used in asynchronous distributed systems): bus and data synchronization handles asymmetric faults in communication activities while voting (and accepting the median value) handles asymmetric values in functional outputs. Temporal asymmetric faults are tolerated while functional ones are excluded from participation in further decision-making. The control actuation outputs are selected as the median of the three computers. The computers are not synchronized but exchange information to consolidate system state to equalize critical variables. The system is designed so that the probability of losing one bus should be no greater than 10^{-5} per flight hour, that of losing two buses should be no greater than 10^{-9} per flight hour and that of losing all three buses should be no greater than 10^{-11} per flight hour.

Employing design diversity to protect against common-mode and common-area failures, each channel is hosted by dissimilar microprocessors, and is physically and electrically isolated from the other two channels. Academic research has demonstrated the system requirements as a potential single point of failure [7]. Rather than have separate software coding teams, Boeing used aggressive fault-intolerant

techniques to develop the requirements. Three different Ada compilers were used to generate the flight software control code from a single source.

6. Observations and Trends

Commercial off-the-shelf components. Increasingly, both the space and avionics domains have increased their usage of open standards and commercial off-the-shelf (COTS) components. COTS products (e.g., microprocessors, middleware, real-time operating systems) are considered to be cost-effective and to facilitate systems-of-systems integration across heterogeneous sub-systems, platforms and vendors. The push towards "faster, better, cheaper" missions, with its attendant emphasis on architectural/design/artifact reuse across missions and product-lines, has led to the popularity of COTS products in domains that have traditionally relied upon custom software and hardware. This is increasingly true of the space [1], the military avionics [14] and the commercial avionics [1] domains – each of the three domains has tended to follow commercial trends in architecture (e.g., ISA, use of Ethernet), but deploys/integrates these COTS products into its respective systems with accommodations for the environment, longevity and safety issues that are characteristic of that domain.

Unfortunately, current COTS standards lack some of the stringent dependability, reaction times and autonomy guarantees required for mission-critical applications. For example, in combat aircraft, COTS products are considered to be an impediment to safety in terrain-avoidance [9] because combat pilots intentionally operate their aircraft close to their physical limits when they fly low-altitude, high-speed maneuvers in hostile environments. On the other hand, COTS products are more suitable for the more stable missions of civilian aircraft and spacecraft where systems are intended to correct for momentary losses of situational awareness on safe, predictable routes that are controlled in every phase of flight and that give pilots or ground control enough warning to recover the craft (with minimum impact on passengers, in the case of commercial aircraft). However, this does not mean that cost-effective COTS products are as yet sufficiently dependable for these types of missions.

Thus, the advent of COTS presents trade-offs between reliability and affordability [18]. There are also other issues with COTS products – obsolescence, updates, integration, validation, and adequate technical support – all of which are also significant considerations. In particular, COTS is not synonymous with simplicity – in fact, the integration of diverse COTS components to form larger, heterogeneous, distributed systems can often lead to emergent complexity that none of the component vendors account for in their product development.

"By-wire" software control and autonomy. Software is becoming an increasing aspect [6] of these systems (see Figure 3 for a trend of software size in spacecraft) – while the Cassini spacecraft might contain only 32,000 lines of code, the International Space Station contains 2 million lines of code running

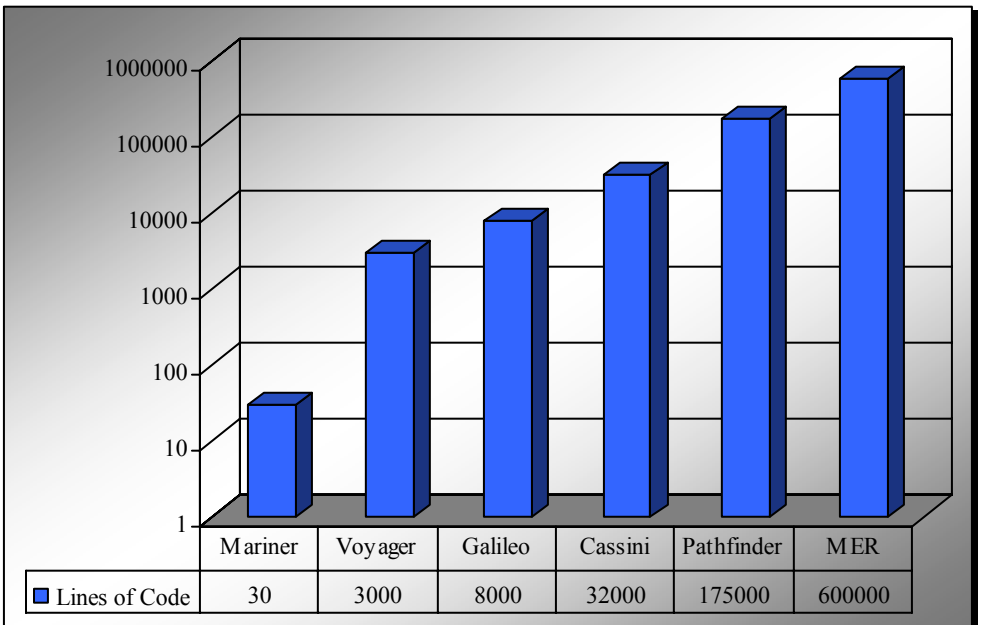

Figure 3: Size of software in spacecraft missions.

on over 50 computers. NASA's missions [22] are increasingly using larger, more complex, more distributed software systems to perform ever more challenging and ambitious missions that must operate for longer durations and survive under more extreme environmental conditions. Also, given that these missions are reaching further into space beyond our neighboring planets or the Moon, additional complexity results because of the autonomy required of the spacecraft because ground control from earth and humans in the loop cannot be expected to make time-critical decisions. The trend in civilian aircraft has also been towards increased autonomous software control of the aircraft, through "fly-by-wire" or electronic flight control systems. Since civilian aircraft carry human passengers (as do manned space missions) they must be certified to meet stringent safety requirements – federal guidelines such as DOA 178B mandate the process and specifications. In fact, every COTS product used in an aircraft must undergo rigorous certification, which is not necessarily the case with the use of COTS software in autonomous, unmanned spacecraft.

Most of the traditional spacecraft mission-assurance tended to focus on hardware fault-protection, e.g., the Mariner system had a backup control system in hardware. NASA's Preferred Reliability and Maintainability Practices [16] reveal the focus on hardware: "fault protection design maximizes the probability of spacecraft mission success by avoiding possible single failure points through the use of autonomous, short-term compensation for failed hardware." [17]

With increasing mission functionality is moving into software (e.g., in the Mars Exploration Rover, the entry, descent and landing were driven by software modules) and more processors being used to control these large systems, software has thus become both an enabling technology and a source of risk and failures. With the control software becoming more concurrent, more distributed and more autonomous, the resulting aircraft and spacecraft systems end up being complex and difficult to make reliable. Therefore, mission-assurance strategies should focus equally, if not more, on software fault-tolerance techniques [26] and also on distributed fault-tolerance principles. This has been recognized by NASA [22] and the avionics industry through increased software verification and reliability practices, e.g. in the Mission Data System [20], a multi-mission framework for building, testing and using software.

Escalating fault sources and evolving redundancy. Table 2 provides an overview of the fault-tolerance mechanisms that we have described for the specific spacecraft and aircraft discussed in this paper. Of particular interest is the evolution of the redundancy strategies in the last column of this table.

Both the spacecraft and civilian aircraft architectures have increasingly come to handle advanced sources of faults by providing for more advanced fault tolerance strategies. The simplest of these strategies, which early spacecraft employed, involved a command-monitor architecture, with one unit issuing all of the commands while the other's purpose was a more passive role, to monitor the command unit; in some cases, a reference/sanity check was provided in order to serve as a basis for comparison in case the two units differed in their observations/values. Another version of this was a primary-backup architecture, which was very similar but for the fact that the monitor, being functionally identical to the command unit, could actually take over seamlessly if the command unit were to fail.

Table 2: Fault tolerance mechanisms for specific spacecraft and aircraft.

Mission/System	**Inception**	**Configuration (Lines of Code, Memory, Hardware, OS, Middleware, Language)**	**Fault-tolerance mechanisms**
Voyager – outer planet flyby	1977-1989	3000 lines of code	Active/standby block redundancy as command/monitor pair
Galileo – Jupiter orbiter and probe	1989	8000 lines of code	Active/standby block redundancy, microprocessor multicomputer
Cassini-Huygens - Saturn orbiter and probe	1997-2005	32,000 lines of code Code written in Ada MIL-STD-1553B Bus (internal redundant bus media)	No single point of failure Primary/backup redundancy Priority-based one-at-a-time handling of multiple simultaneous faults $3.26 B
Mars Pathfinder - Mars lander and rover	1996-1997	175,000 lines of code 32-bit RSC-6000 processor 128MB DRAM VME backplane VxWorks real-time OS Object-oriented design (in C) Special "point-to-point" MIL-STD-1553B Bus	Selective (not full) redundancy Complete environmental testing Adoption of vendor's QA practices Based on short mission duration, budget cap and extreme thermal/landing conditions $280 M
Airbus A340 – flight control computer	1993	Two different processors (PRIM and SEC)	Design diversity emphasized to handle common-mode and common-area failures
Boeing 777 - flight control computer		Code written in Ada ARINC 629 bus Dissimilar multiprocessors	Triple-triple modular redundancy for the primary flight computers Goal to handle Byzantine failures, common-mode and common-area failures Physical and electrical isolation of replicas

In both the command-monitor and the primary-backup strategies, there was inevitably a system "pause" when the reconfiguration occurred (i.e., when the primary/command unit failed). To work around this, and to provide timely fault-recovery, active or hot-standby replication was adopted for the more time-critical activities and subsystems. In this case, if a unit is replicated, its (two or more) replicas would remain identical at all times by operating in lock-step; the advantage was one of failure masking should any one of the replicas fail; the disadvantage was the use of additional resources and computing power. In spacecraft, this is very rarely done, except for the most time-critical of units or for some units in time-critical phases. All of these strategies protect against single crash-faults or single communication-faults, assuming the independence of fault sources. This can be carried one step further to triple modular redundancy (TMR) which protects against asymmetric faults by voting or median picking of the outputs of the simultaneously operating replicas in order to produce a single output of high integrity.

Mission-assurance for traditional spacecraft missions has primarily focused on complete primary-backup or command-monitor redundancy to avoid single points of failure. With NASA's increased focus on "faster, better, cheaper" missions, strategic, rather than complete, redundancy has become an attractive goal for the smaller missions that are increasingly the norm. One of the distinct differences of the civilian aircraft domain, as compared with the space domain, is a significant emphasis on design diversity [30] to protect against both common-mode and common-area failures, by using dissimilar processors, different software versions, and distinct physical locations for deployment of replicas. In terms of fault protection, aircraft architectures tend to employ block level replication in command/monitor pairs, triple modular redundancy, and up to triple-triple modular redundancy (as compared with dual redundancy in some parts of spacecraft architectures). We anticipate that aircraft will evolve their focus to include security attacks as well in the future, as a part of their overall emphasis on safety and dependability; on the other hand, spacecraft will continue to focus on availability and longevity, unless the missions are manned, in which case safety becomes the primary consideration.

Domain-specific observations. Fault-protection during space missions has typically incorporated "spacecraft safing" reaction to runtime anomalies or failures. Safing involves reporting error messages to ground control, ceasing normal spacecraft operations, orienting the spacecraft's solar panels towards the sun, and awaiting command sequences from earth. While this has been successful for previous missions and is useful for problem diagnosis, shutting down normal operation is clearly not feasible when the spacecraft undergoes critical events, such as orbit insertion; thus, fault-protection was effectively disabled during such critical phases to ensure that normal operation was not overridden. Secondly, the aggressive approach of safing might clearly not be feasible for relatively minor anomalies since it effectively halts scientific exploration activities as well. Thirdly, safing might not be appropriate for missions where the spacecraft needs some level of independence to carry out its operations since its time-critical activities can simply not tolerate the large round-trip latencies associated with commands from ground control. Thus, the need for autonomy and the scope of the mission (moving further away from earth) might increasingly lead to a departure from the safing response to all kinds of anomalies.

Commercial software projects have project-delivery deadlines that are dictated by market pressured, and missed deadlines have adverse financial/business consequences; in space applications, the timeline for the completion of a spacecraft-software project is linked to the precise, relative locations of the earth and the orbiting/landing/exploration targets for the mission and is, therefore, governed by the laws of celestial mechanics. Thus, for a successful mission, there is often no alternative (except for costly delays or abandoning the project) other than to make the stipulated project deadline on time.

6. Conclusions

In this paper, we have attempted to survey, compare and contrast the different architectural techniques that have been used to improve system reliability in spacecraft and civilian aircraft. The generic fault-protection strategies that we discuss are substantiated by concrete examples taken from real missions/systems, including spacecraft (Voyager, Galileo, Cassini-Huygens, Mars Pathfinder) and civilian aircraft (Airbus A330/A340/A380 and Boeing 777). Using these fault-tolerant architectures as representative case studies of the space and avionics domains, the paper concludes by observing trends and projecting future developments in these domains.

Acknowledgements

This material is based upon work performed by Daniel P. Siewiorek supported by a grant from the Office of Naval Research, Interoperability of Future Information Systems through Context-and Model-based Adaptation (contract N00014-02-1-0499), the Defense Advanced Research Projects Agency under contract number NBCHD030010, the National Science Foundation under grant numbers 0205266 and 0203448 and the Army Research Office grant number DAAD19-01-1-0646. Priya Narasimhan was supported by NSF CAREER Award CCR-0238381, the Army Research Office grant number DAAD19-01-1-0646, the Army Research Office grant number DAAD19-02-1-0389 (”Perpetually Available and Secure Information Systems”) to the Center for Computer and Communications Security at Carnegie Mellon University. The authors gratefully acknowledge Savio N. Chau of JPL for his invaluable feedback on this paper. The authors also thank Laura Forsyth for her help in preparing this paper.

References

[1] C. Adams, “COTS operating systems: Boarding the Boeing 787,” *Aviation Today*, July 2005.

[2] L. Alkalai, A. Tai and S. Chau**, “**COTS-based fault tolerance in deep space: Qualitative and quantitative analyses of a bus network architecture*,” IEEE International Symposium on High Assurance Systems Engineering,* November 1999, Washington D.C., pp. 97-104.

[3] D. Briere and P. Traverse, “Airbus A320/A330/A340 electrical flight controls: A family of fault-tolerant systems,” *International Symposium on Fault-Tolerant Computing*, Toulouse, France, June 1993, pp. 616-623.

[4] J. Clawson, “Mars Pathfinder “common sense” mission assurance,” *Aerospace Conference*, vol. 5, March 1998, pp. 477 – 489.

[5] H. Hassan and J. C. Jones, “The Huygens probe,” European Space Agency no. 92, November 1997.

[6] G. Holzmann, “Developing reliable software for space missions,” *International Space Development Conference*, Washington D. C., May 2005.

[7] J. C. Knight and N. G. Leveson, “An Experimental Evaluation of the Assumption of Independence in Multi-version Programming,” *IEEE Transactions on Software Engineering*, vol. SE-12, no. 1, January 1986, pp. 96-109.

[8] R. W. Kocsis, “Galileo Orbiter Fault Protection System, Jet Propulsion Laboratory, California Institute of Technology.

[9] P. A. Jackson, D. Bostich and J. Padukiewicz, “Terrain avoidance: When COTS won’t do,” *Avionics Magazine*, June 2005.

[10] C.P. Jones, “Automatic Fault Protection in the Voyager Spacecraft.” Jet Propulsion Laboratory, California Institute of Technology, AIAA Paper No. 79-1919, Pasadena, CA, 1979.

[11] C. P. Jones and M. Landano, “The Galileo Spacecraft Design,” AIAA Paper No. 83-0097, *Aerospace Services Conference*, Reno, NV, January 1983.

[12] J. H. Lala and A. L. Benjamin**, "**Advanced fault-tolerant computing for future manned space missions," *AIAA/IEEE Digital Avionics Systems Conference*, Irvine, CA, October 1997, vol. 2, pp. 8.5-26—8.5-32.
[13] R. C. Lisk, "NASA preferred reliability-practices for design and test," *IEEE Reliability and Maintainability Symposium*, January 1992, pp. 7-11.
[14] M. K. J. Milligan, "Implementing COTS open systems technology on AWACS," *Crosstalk: The Journal of Defense Software Engineering*, September 2000.
[15] B. K. Muirhead, "Mars Pathfinder flight system design and implementation," *Aerospace Applications Conference*, vol. 2, February 1996, pp. 159-171.
[16] National Aeronautics and Space Administration, *NASA Preferred Reliability and Maintainability Practices*, http://www.hq.nasa.gov/office/codeq/rm/prefprac.htm
[17] National Aeronautics and Space Administration, *Fault Protection*, Preferred Practice No. PD-EC-1243, October 1995.
[18] A. Nikora and N. Schneidewind*, "*Issues and methods for assessing COTS reliability, maintainability and availability," *COTS Workshop, International Conference on Software Engineering*, Los Angeles, CA, May 1999.
[19] E. C. Ong and N. Leveson, "Fault-protection in a component-based spacecraft architecture," *International Conference on Space Mission Challenges for Information Technology*, Pasadena, CA, July 2003.
[20] R. D. Rasmussen, "Goal-based fault tolerance for space systems using the Mission Data System," *IEEE Aerospace Conference*, Big Sky, MT, March 2001, pp. 2401-2410.
[21] R. Ramesham, "Extreme temperature thermal cycling tests and results to assess reliability for Mars Rover flight qualification," *Microelectronics Reliability and Qualification Workshop*, February 2004.
[22] P. Reagan and S. Hamilton, "NASA's mission reliable," *IEEE Computer*, January 2004, pp. 59-68.
[23] D. Siewiorek and R. Swarz, Reliable Computer Systems: Design and Evaluation, Third Edition, A. K. Peters, Ltd., 1998.
[24] J. P. Slonski, "System fault protection design for the Cassini spacecraft," *IEEE Aerospace Applications Conference*, vol. 1, 1996, New York, NY, pp. 279-292.
[25] N. Storey, *Safety Critical Systems*, Addison Wesley, 1996.
[26] W. Torres-Pomales, "Software fault tolerance: A tutorial," *NASA Technical Report NASA/TM-2000-210616*, October 2000.
[27] P. Traverse, I. Lacaze and J. Souyris, "Airbus fly-by-wire: A total approach to dependability," *IFIP World Computer Congress*, Toulouse, France, August 2004.
[28] D. F. Woerner and D. H. Lehman, **"**"Faster, better, cheaper" technologies used in the attitude and information management subsystem for the Mars Pathfinder mission**,"** *Aerospace Applications Conference*, vol. 2, February 1995, 155-167.
[29] Y. C. Yeh, "Safety critical avionics for the 777 primary flight controls system," *IEEE Conference on Digital Avionics Systems*, vol. 1, Daytona Beach, FL, October 2001, pp. 1-11.
[30] Y. C. Yeh, "Unique dependability issues for commercial airplane fly-by-wire systems," *IFIP World Computer Congress*, Toulouse, France, August 2004.