JOURNAL OF AEROSPACE COMPUTING, INFORMATION, AND COMMUNICATION
Vol. 3, March 2006

A Framework for Robust and Flexible Ground Station Networks

James Cutler and Armando Fox
Stanford University, Stanford, California 94306

Space communication systems are plagued by complexity resulting in limited access to orbiting satellites and high mission operation costs that ultimately reduce mission yields and capabilities. In particular, ground stations, the access point between space and terrestrial networks, suffer from monolithic designs, narrow interfaces, and reliability issues that raise significant engineering barriers. We have reduced these barriers by decomposing monolithic ground station functions into basic component services and virtualizing their software and hardware interfaces. We captured these core services in a command and control protocol, called the Ground Station Markup Language. This modularization has also enabled us to apply recovery-oriented computing techniques, specifically recursive restarts, to quickly recover from software failures and improve overall ground station availability. Our testbed consists of globally distributed ground stations supporting university satellite missions. With case studies, we demonstrate how our work enables rapid prototyping and reconfiguration within a ground station and how it facilitates coordination of networked, heterogeneous ground stations by hiding low-level implementation details.

I. Introduction

A STATED goal of the space operations research community is the removal of communication as a constraint for space missions.[9] Two of the space communication constraints impeding missions are limited access to space networks and decreased reliability. As we discuss in the next section, current ground station networks do not provide cost effective 24/7 connectivity to satellites. Existing ground-based networks do not have the antenna infrastructure to support 24/7, on-demand access, and their services are often beyond the financial grasp of many small satellite missions. In addition, fault-free systems will continue to be elusive as these systems become more complex and distributed. As space systems adopt Internet technologies to meet their networking needs, we expect similar production system faults as those found in existing Internet systems to be present as well.[35,36,42,46,52,53,57]

Others working on improving access to space systems have agreed on the value of standardization as a means of increasing interoperability.[1,8] Similarly, there is widespread movement toward rapid prototyping and deploying inexpensive space missions by using commercial-off-the-shelf (COTS) components.[8,10] Our goal is to advance both trends and add the following important contribution: for these heterogeneous COTS-based systems to remain both affordable and robust, we propose to apply techniques of recovery-oriented computing (ROC),[54] which enable the construction of reliable Internet services from unreliable COTS parts that were not designed to work together. The ROC philosophy assumes failures are a fact and emphasizes fast recovery over solely fault avoidance.

Two avenues of our research have focused on making it easier to "access satellite space science data as easily as one would a web server".[43] First, based on our extensive experience building, launching, and operating small satellite missions, we have identified a suitable functional decomposition for satellite ground station networks into modular, layered services. The resulting modularization has allowed us to leverage Internet-based tools for monitoring system health and accessing ground station services. And, equally important, it has allowed us to apply ROC techniques

Received 11 January 2005; revision received 11 January 2005; accepted for publication 27 April 2005.
*Email: jwc@stanford.edu, fox@cs.stanford.edu

to improve system availability. Our second avenue of research has captured this decomposition of ground station services and mission products in an extensible representation called the Ground Station Markup Language (GSML). Because GSML is portable across heterogeneous ground station installations, it greatly increases the degree of collaboration possible between these stations.

Our work has increased reliable ground station coverage for university and small satellite missions. Our GSML reference implementation, Mercury,[25] is being used by several universities to prototype a ground station network that harnesses idle ground station resources and provides global ground station coverage. GSML has enabled independent development of commodity ground station services and a framework for extending these services to commodity satellite operations tasks.[22,31] Our application of recursive restarts,[6] a ROC high-availability technique, has reduced recovery time by a factor of four in Mercury systems and eliminated the adverse effects of many ground station software failure modes. Mercury support of a scientific small satellite mission, QuakeSat-1,[14,24] has provided low-cost flexible satellite contact opportunities that currently do not exist among commercial providers.

This paper provides an overview of our modularization efforts and two case studies that measure their impact on space missions. First, we discuss our satellite and Internet background experiences and the design principles we have extracted from them. Then, we develop an informal reference model that deconstructs monolithic ground station capabilities into modular, layered services and capture this model in a portable markup language. Next, we extend the core ground station services to include a mechanism for flexible application-level support through the use of virtual machines. We demonstrate how GSML and our reference implementation have increased satellite contact capabilities through the cooperation of an international university ground station network. We also demonstrate how this modularization enabled the application of a high-availability technique based on recursive restarts to improve ground station availability. Our final case study measures the impact of our work on enabling extremely low-cost experimental satellite missions. We conclude with related and future work.

II. Background

Through nine recent Stanford satellite missions, we have experienced space mission trends firsthand.[21,24,44,60] Most notably, the Opal mission demonstrated that low-cost, albeit high-risk, university satellites offer an excellent platform for experimental testbeds.[29,44] Opal's primary mission was to explore mothership/daughtership mission architectures for distributed sensing applications. Six daughterships, called picosatellites, were developed by three independent teams from the Aerospace Corporation, Santa Clara University, and AMSAT. Opal was built primarily with non-hardened COTS components. It was launched in January of 2000 and operated for two and half years. Once the Opal picosatellites were deployed and the mission settled into routine operations, the Opal operations team focused on enhancing the mission operation systems through Internet-based operational methods.[12] Our goal was to make accessing Opal as easy as we access web pages and with all the attendant benefits.

The mothership/daughtership innovation of the Opal mission has been extended into a global spacecraft form factor specification called CubeSat that defines modular spacecraft structural units of 10cm cubes and a standardized launcher mechanism.[26] With the introduction of this standard in 2000, the community of university satellite builders has grown from roughly 20 to over 70 institutions. The first six CubeSats were launched in 2003 and 18 more are scheduled for an early 2005 launch.[11]

A. Problem Statement

Through these satellite mission experiences, we have first-hand experience with two constraints facing space communication systems: lack of cost-effective access networks and a decrease in network reliability as space systems become more complex.

1. Lack of Access

A trend in future space missions is on-demand, 24/7 access to orbiting satellites.[2] For extremely low-cost, experimental space missions,* this type of access coverage is not affordable. First, it is cost prohibitive for these small missions to build their own ground stations let alone an entire network that provides global coverage. Typical

* For an example, see QuakeSat-1[14,24] which was developed for under US$1M. We describe this mission in more detail in a case study found in section V.

industry ground stations capable of megabit per second downlinks have extensive specialized hardware components and, on average, cost several hundred thousand dollars. This does not include maintenance requirements or the overhead of globally distributed ground station sites. Second, existing access networks are not affordable for these experimental satellite missions. During development of QuakeSat-1, a commercial ground station network provider requested US$150K for integration fees and US$550 per pass for support. Six months of operation with six passes a day would then cost US$744K, or three fourths of QuakeSat-1's US$1M budget. Clearly, for missions such as QuakeSat-1, these access network services are not affordable.

The high cost of legacy ground station systems can partially be attributed to the complexity of traditional space systems and their long-term nature (3-5 years of development and 3-10 years of on-orbit operation). These missions tend to be monolithic stovepipes; the developed systems are custom, one-time solutions optimized for a particular mission where COTS solutions are avoided because of obsolescence issues. This results in components that are difficult to share with other missions since flexibility was not a requirement for the original mission. In the case of ground stations, it is not uncommon for costly hardware additions to enable multiple mission support.[32,34] Also, since flexible multi-mission support was not a requirement for these ground stations, they lack standard interfaces for their services. Those that exist cover only a small set of possible ground station interfaces and services.[28]

2. Decreased Reliability

During our development of Internet-based operations platforms (which included deployment of web servers, file servers, small Linux-based clusters, databases) we discovered that set against the potential benefits of COTS-based systems are challenges as well. Most notably, and the one we tackle in this work, is the integration of non-mission-critical components whose behaviors were not fully understood and typically designed for lighter workloads into mission-critical system service. Hardware fails (such as power supplies and disk drives) and software locks up or fails to free resources forcing system reboots. Our experience is that partial failures are to be expected and the system should mask them and operate gracefully.

Additionally, the complexity of space systems is increasing and with it the possibility for systems errors and failures. Rising levels of automation introduce more software code and more computer hardware. The global distribution of space system components requires extensive networking. Increasing levels of international collaboration and peer-to-peer space systems result in complex interfaces and interactions that are difficult to model. Space systems only become more complex as they increase in capability. As this complexity increases, fault-free software will continue to be elusive. System engineers and researchers have traditionally relied on fault avoidance techniques, system validation, and analytical models to try and build fault-free software. These decades of experience have led to numerous valuable techniques,[49,55,58] which have improved the reliability of our systems, but these same efforts have also demonstrated that accurate modeling and/or verification of complex computer systems and their environment is impractical in most cases. Therefore, as space system complexity increases, component failures are to be expected and we must cope with them.

B. Design Guidelines

Our vision for tackling these barriers, the lack of access and the increasing unreliability of complex systems, is to network enable independent, globally distributed ground stations, to enable development of commodity services for these ground station networks, and to provide high-availability techniques that enable reliable systems from unreliable components. To this end, we have extracted several design guidelines from our small satellite and Internet-based operations experience for implementing our vision.[43]

Provide Layered Control of Complex Systems—Stanford's automated tracking capabilities were insufficient for one of Opal's picosatellites, the Aerospace tethered satellites, due to their small size and fine pointing requirements of our 45m ground station dish. Thus, Aerospace developed novel spiral tracking methods to hunt for their picosatellites.[27] This demonstrates that while it is highly desirable to automate routine tasks such as antenna pointing, it is necessary to expose lower-level primitives to enable customization of ground station services. Without this ability, the Aerospace mission would not have been a success. Our research efforts provide a structured, principled method for both adding low-level customization and maintaining higher-level automation, for without both of these, it will be difficult to adopt standard ground station interfaces. We use a self-describing format based on XML to enable extension without

breaking backward compatibility. We expect this to improve multi-mission support of ground stations and increase overall accessibility to space.

Explicitly Encapsulate State—Opal's communication system used terminal node controllers (TNCs) on board and in the ground station to provide modulation, demodulation, frame synchronization, and network protocol services. State parameters associated with communication links were hidden within the TNCs and could not be extracted by outside components. Since the state was inaccessible, TNC failures resulted in the loss of communication and data. Separation of this state and the ability to export it would have allowed additional, independent mechanisms to protect this state and enable fast recovery of failed TNCs. Thus, mission critical state should be identified so that adequate failure recovery strategies can be deployed. Our work extracts this state for ground station network systems and provides a markup language for explicitly encapsulating this state.

Separate Data Format from Implementation—Our mission and ground station control systems use protocols associated with the Web (HTTP, TCP, SSL) to transfer control and data messages. Our COTS Web clients and servers all understand the lingua franca of the Web, HTTP, the only "contract" between Web entities. HTTP separates data format of the exchanged data from the implementation entities that operate on the data. This enables clients and servers to be written in a variety of languages, the choice usually being determined by the engineering demands of the specific component. The self-describing extensible nature of HTTP also enables backward compatibility: if the Web server updates its functionality, it does not need to notify older Web browsers in advance in order for them to continue to use the server. In contrast, traditional object-oriented software systems, such as those built using CORBA,[61] rely on a more heavyweight calling protocol, require substantial prior agreement on data types and calling conventions, and do not generally tolerate independent component evolution without strict versioning. In an effort to promote this same type of implementation flexibility and backward compatibility, we are developing a portable, extensible, self-describing framework for commanding and controlling ground station services. Our implementation is based on XML and is called the Ground Station Markup Language (GSML).

Preserve Fault Isolation Boundaries—Strong isolation boundaries between Opal subsystems prevented failures from propagating to other components. At one point, during on-orbit operation, battery systems reached an elevated temperature of 80C, which is well beyond their 35C specification (the point at which they explode catastrophically). Fortunately, these batteries were isolated in sealed aluminum boxes and potted in epoxy. This prevented any adverse effects from spreading and enabled the mission to operate for another two years. Similarly, high levels of fault isolation—both software and hardware—are needed as we build ground station systems from COTS-based components which are often themselves complex systems whose behaviors are not fully understood and which can fail in unknown ways. Our goal is to structure overall ground station applications so that the failure of one component does not necessarily result in loss of service.

Design for Restartability—Most of the software failures experienced in our ground systems are cured by restarts, where rebooting CPU's or restarting failed processes restored the system to an operational state. This has also been the experience of other Internet services.[39] Thus, we are developing a framework that provides recursive restartability (RR), which we define as the ability of a system to tolerate restarts at multiple levels (e.g. thread, process, node, system). We have argued that RR improves failure tolerance[41] through a combination of failure-triggered reactive component restarts and proactive restarts on components vulnerable to software aging[47] that mask many faults from the outside world, thus making the system as a whole more fault tolerant. Our application of two previous guidelines (state encapsulation and isolation boundaries) allows us to apply RR to our ground station infrastructure; mission critical state is captured in dedicated state stores that are protected during restarts and isolation help prevents error propagation among components. We demonstrate how RR improves system availability and sometimes completely eliminated the effects of failures during satellite contacts.

III. Ground Station Decomposition

This section describes our application of the first three guidelines from section II.B to ground station networks. We have decomposed ground station network functionality into core services and described them in an informal

reference model. We then capture this model in a command and control language built on an XML framework, called the Ground Station Markup Language (GSML). We also extend core ground station services with a mechanism that combines recent advances in software defined radios and virtual machines to reduce the complexity of specialized hardware components and enable generic application level support in ground stations.

We draw additional insight for these modularization efforts from the use of the Internet Protocol (IP),[18] a global standard for datagram delivery. IP's strength lies in the ability to layer additional services above it with increased capabilities, such as advanced networking protocols (TCP, RTP) and application level services (email, file transfer). The result of this layering and modularization is that the Internet has proven extremely fertile ground for explosive, innovate growth and rapid prototyping. HTTP,[48] for example, was originally designed for simple document transfers and now carries diverse traffic such as real-time stock trading from cell phones. Additionally, extensive commodity solutions are available for HTTP systems. This fertile environment has even allowed graduate students to create systems that result in new English verbs such as "google". In light of successful Internet systems, we are led to ask several questions about rearchitecting ground station services:

- Can a ground station network be decomposed into basic building blocks that enable custom composition of services by the satellite operations teams conducting satellite passes?
- Does this decomposition then enable and promote development of interoperable COTS-based implementations of ground station services?
- As was the case with Internet applications, can we now apply high-availability techniques to ground station systems?

A. Reference Model

We begin with a decomposition of typical ground station (GS) functionality into core, layered services which we have captured in an informal reference model.[25] The model is hierarchical and layered according to levels of autonomy; lower levels capture hardware devices and higher levels capture autonomous ground station services. It defines ground station hardware and software objects, such as antennas and satellite contact session descriptions, and their associated state attributes. It also captures services and functions that can be performed with ground station objects.

Virtualization of lower layers beneath higher level services provides several benefits. Lower level implementation details are masked from higher layers. Access control policies are enforceable at layer boundaries. Layering also enables local optimization of lower level control. During the first several weeks of operation, Opal contacts were conducted at the SRI 45m radio telescope at Stanford.[13] The Opal operations team tasked the dish through a higher level interface. We provided the dish operator with a start time, an end time, and Keplerian element sets that were used to calculate Opal's position. This higher level interface masked low-level details from the inexperienced Opal operations team and prevented inadvertent errors.

Of course, in some applications layer violation is expected. Local automation of the normal case of lower level services may not adequately meet mission needs or desires. Lower level access that bypasses higher services should be possible for trusted users. For example, consider again the SRI 45m dish and the tracking of the picosatellites. Due to the margin of error in the Keplerian element sets of the picosatellites, the local tracking algorithm used for Opal was inadequate to track the small picosatellites. Engineers from the Aerospace Corporation bypassed the standard auto-tracking software and utilized novel spiral methods to acquire their picosatellites.[27]

We describe the three levels of our reference model below. The hardware level captures the fundamental capabilities of low-level ground station hardware and enables generic commanding of heterogeneous hardware components. The session level captures typical automation tasks and services of a ground station. Finally, the network level captures the services of a federated ground station network.

1. Hardware Level

The hardware level (HL) captures the fundamental capabilities of low-level ground station hardware and enables generic commanding of heterogeneous hardware components. Device-specific commanding protocols have been abstracted away to present a standard interface to ground station hardware (comparable to device drivers found in computer operating systems). Control of the station at this level is a master/slave paradigm with no autonomy in

the GS; all commanding is done from remote, master operators (or their software). Antennas, low-noise amplifiers, heaters, and CPU's are example hardware objects.

In an effort to simplify the conceptualization of ground station capabilities, we have created a virtual object, called a *pipeline*, which describes the fundamental function of a ground station, the bridging of space and ground networks through wireless communication channels. Pipelines capture the fundamental characteristics of the channel while masking low level implementation details (such as hardware configuration and multiplexer pathways). This allows ground station administrators to define pipelines that specify valid hardware configurations and mask the complexity of implementation details from the end user. Receive pipelines convert space radio transmissions into digital bits that can be placed on terrestrial networks, while transmit pipelines perform the inverse of this, and transceive pipelines do both. Figure 1 is a block diagram of example pipeline components. Typical pipeline descriptions contain information such as location, antenna receive gain, frequency ranges, and data handling functions.

2. *Session Level*

The session level (SL) captures typical automation services of a single ground station installation. Users task a ground station to maintain a contact session with a satellite and employ local automation services of the station to perform routine component control during the pass. At the heart of this level is the *session* object which the ground station "executes" on a reserved set of hardware over a specific interval of time to maintain communication with a target satellite. The session object parameters include the start time, end time, list of reserved pipelines, targeted satellite, and owner of the session. Additional SL objects include the session queue which maintains a list of session objects, the session product that contains an archive of station telemetry and communication channel data of executed sessions, and ground station users.

SL services automate typical ground station services such as antenna tracking, Doppler shift radio frequency correction, satellite ranging, and position estimation. These services employ hardware-level primitives to control lower level ground station resources. Users can bypass these automation services to directly control all or part of station hardware to best fit their mission requirements. In the case of full user control of lower level commands, a session is still scheduled to reserve ground station pipeline components.

3. *Network Level*

The network level (NL) captures the services of a federated ground station network (FGN).[43] A FGN is a collection of autonomous, globally distributed ground stations with network services such as authentication, resource discovery, and scheduling. The underlying motivation for such a network is the sharing of geographically diverse, heterogeneous ground stations and the synergy of cooperating ground station teams to enhance satellite contact sessions.

A FGN preserves the primary function of ground stations, to provide connectivity between space and ground networks. As such, the basic building blocks of an FGN are globally distributed ground station installations under

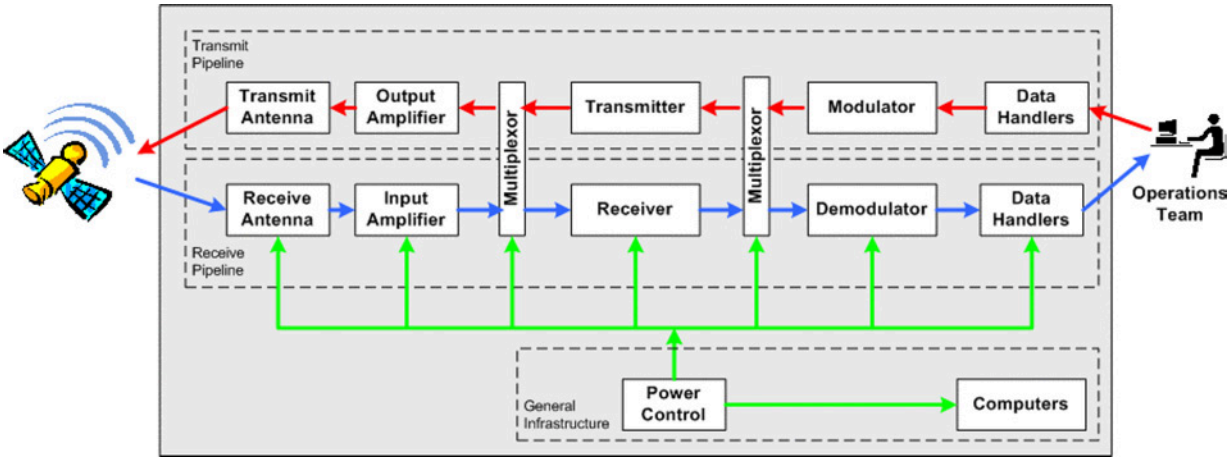


Fig. 1 Block diagram of ground station pipelines.

different administrative domains; they are owned and operated by different institutions. To preserve the autonomy of FGN members, stations dynamically join and leave the federation based on local constraints, such as routine maintenance, failure repair, or security reasons. Individual user access control is at the discretion of local administrators. It is not a requirement for stations to be identical at the physical level; heterogeneity is masked by the standardized interfaces. Station capabilities (such as frequency ranges, antenna gain, and data services) are captured in a ground station specification that enables FGN users to search for compatible stations. The FGN concept fundamentally alters mission-specific ground stations by harnessing their services and creating a system of network-accessible stations that provide greater coverage to multiple missions.

B. GSML

We have captured this reference model in markup language called the Ground Station Markup Language (GSML) which provides portable command and control of ground station services.[16] Our goal is the acceptance of a layered, standard approach to architecting ground station systems and not setting GSML as a specific standard. As such, GSML is currently a research tool for exploring the proper layering for ground stations as well as a pedagogical tool for teaching ground station systems. GSML uses the extensible markup language (XML) framework,[33] and is specified in XML Schema 1.0.[62] XML was chosen because of its inherent extensibility and wide adoption among Internet services resulting in a variety of COTS tools such as document generators, document checkers, and XML parsers for most modern programming languages.

GSML consists of objects, methods, and a protocol to describe interaction with and among these objects. *Objects* describe FGN components such as users, stations, and antennas. They capture state and attributes that identify and describe FGN components. *Methods* capture capabilities and services of FGN objects such as scheduling contact sessions and commanding antenna motion. *Protocols* describe procedures for FGN component interactions and are captured in prose text versus the XML Schema specification. An example session level GSML message that schedules a contact session is shown in Fig. 2.

A draw back of GSML is its lack of bit efficiency; it is quite verbose, as is the nature of markup languages. Customized formats, such as processor microcodes, could be used to improve the efficiency but this severely restricts the extensibility of GSML, as well as the use of COTS tools. Given that we have deployed an entire GSML-enabled, single station control system (web server, database, Java back-end control software) on an aging legacy PC (Intel

```
<GSML SRC="QuakeSatOps" DST="SSDL-Durand-GS">
  <ADDSESSION>
    <SESSION>
      <STARTTIME>
        <TIME>
          <SECONDS-1970>1071698270</SECONDS-1970>
        </TIME>
      </STARTTIME>
      <ENDTIME>
        <TIME>
          <SECONDS-1970>1071699770</SECONDS-1970>
        </TIME>
      </ENDTIME>
      <TARGET>
        <NAME>QuakeSat-1</NAME>
        <ID>27845</ID>
      </TARGET>
      <PIPELINE NAME="TRX-70cm-9600"/>
      <VM NAME="Quake-VM-v01"/>
      <SERVICES>
        <AUTOTUNE TYPE="basic"/>
        <AUTOTRACK TYPE="basic"/>
      </SERVICES>
      <USER NAME="QuakeSatOps"/>
    </SESSION>
  </ADDSESSION>
</GSML>
```

Fig. 2 A prototype GSML-SL message.

Pentium II@200 MHz, 128 MB RAM, and 20 GB hard drive), we expect that Moore's Law will be more than adequate to compensate for the lack of XML efficiency. Compression of messages is also a possibility for improving efficiency, particularly in the case of limited bandwidth connections. On a side note, binary data can be captured in text-based XML messages through binary encoding schemes such as Base64.[23]

C. An Extension to Core Services

We extend core ground stations services with a mechanism that combines recent advances in software-defined radios (SDR) and virtual machines to reduce the complexity of specialized hardware components and enable generic application level support in ground stations. Software defined radio systems are replacing specialized hardware components within ground stations.[34] Current SDRs integrate configurable SRAM-based FPGAs and DSPs with off-the-shelf computing platforms running a variety of operating systems.[4] Development efforts are pushing the flexibility of SDRs even further to replace the specialized FPGAs and DSPs with general purpose processors.[15] Now consider the virtualization technology in modern computer systems that enable multiple instances of guest operating systems, called virtual machines, to run a single host system. We combine these two technologies and develop a ground station architecture where SDRs running on general purpose computers are encapsulated in virtual machines (VMs). This enables us to capture the entire digital manipulation process in a well-defined software container, the VM. Services such as modulation/demodulation, bit synchronization, and forward error correction, which in the past have taken large racks of specialized equipment,[34] are folded into one or more virtual machines running on off the shelf computers with high speed internetworking. From the ground station's perspective, this dramatically simplifies hardware configurations, speeds the transition to network and software-centric systems, and provides greater configuration flexibility.

A variety of possibilities exist for the execution of VMs to perform the digital manipulation services. Ground station administers can predefine their own VMs with a given set of functionality. Our experience with VMs that perform these stateless services (versus long term data storage such as databases) result in VMs that are less than 10 GB in size. With the cost of hard drive storage less than US$0.50 for a gigabyte of storage, hundreds of VMs can be stored for less than a US$1000, which provides a virtually endless possibility of ground station digital manipulation configurations. A second possibility is the execution of mission-specific, user-defined VMs by the ground station. Users can build and customize their own VMs and upload them to ground stations to perform mission-specific services. The lifetime of the VM can be limited to single contact sessions or to span multiple sessions over time. Also, the VM can be isolated at the network level to allow access only to particular GS hardware systems.

As with most engineering solutions, the use of VM-based SDRs does not come without a tradeoff. Dedicated, specialized hardware is faster than software running on general purpose computers. Ground stations running VMs will not boast the maximum speed rates of space communication systems. But given the steady advance in computational power and the desired internetworking goals of modern space systems, we feel it is appropriate to trade off highest speed communication for flexible architectures that cost-effectively support the trends in space networking. Additionally, though the hardware complexity is reduced, the complexity does not disappear and manifests itself as software complexity in the software defined radio services. As such, reliability of these systems is a more pressing concern for us than integration. Our future work will determine if ROC such as RR techniques are applicable in the context of VM-based SDRs.

D. Discussion

As discussed in section II.B, the monolithic culture of legacy space systems had no motivation to produce a "lingua franca" for ground stations and their users. The few existing standards capture a limited spectrum of possible services. The result is a lack of accessible ground station infrastructure where satellite contact opportunities are severely limited. One of our goals is to promote standard access to ground stations and enable flexible, custom composition of their services.

Our informal reference model expands those implemented in current standards to include lower level hardware control primitives and higher level network collaboration services. As a result, the end user has more customization options to meet the specifics of their mission requirements. GSML embodies this model and provides a structured framework for future extensions and modifications of ground stations services. Thus, the reference model and

GSML define the data objects and services of a *virtual ground station* providing a portable format that abstracts away implementation details of heterogeneous ground stations.

By extending core services to include virtual machine execution, we have provided a mechanism for extremely flexible application support at ground stations. We feel the benefits of the VM-based SDRs are numerous and outweigh the decrease in speed described above. Specialized hardware pipelines in ground stations shrink as more capabilities are folded into the VM. The result is simpler ground station hardware and the reduction in the uniqueness of ground stations; fundamental hardware constraints such as antenna gain and frequency bandwidths remain but everything else is encapsulated in software that can be run at any ground station. A second benefit is an increase in the capabilities of hardware-limited ground stations. What once required specialized hardware and sophisticated installation now can be downloaded and executed in a VM. This facilitates upgrades and deployment of new technology. A third benefit is in the customization of GS capabilities by the end users. They provide the VM, one they know works and is specialized for their communication infrastructure. The result is a step toward *software-defined ground stations*, a system where fundamental physical limitations are reduced and the majority of services are performed in software.

In the next sections, we describe several case studies that measure the impact of our work on increasing reliable communication access to satellites.

IV. Case Study—Exploiting Decomposition to Improve Availability

In this section, we discuss the design and development of a GSML reference implementation. We also apply a mechanism for high availability based on recursive restarts.

A. GSML Reference Implementation

We have developed a GSML reference implementation called Mercury[25] that provides mixed initiative control of ground station services over the Internet. The current release, version 1.2.0, supports the hardware and session levels of GSML, and the network level is under alpha testing. Mercury is deployed at two SSDL ground stations operating four transceive pipelines in California and Alaska. Installation is underway at a beta-test group consisting of universities in Norway, Germany, Denmark, and Iowa. At Stanford stations, Mercury is deployed on Linux-based personal computers and Java-enabled microcontrollers. The initial installation was a single PC system (Pentium II, 200 MHz, 256 MB of RAM, RedHat 7.2). Recently, we upgraded to two dedicated servers each with dual 2100 Athlon MP processors and 2GB of RAM running Fedora Core 1.

Mercury is a loosely coupled system of networked software components modeled after a classic three-tiered Internet web service (a web server front-end, application logic captured in middleware, and back-end storage with a database) plus additional back-end support software that conduct satellite contact sessions. Figure 3 is a block diagram of a Mercury ground station system. The master group of components runs continuously serving Internet-initiated requests and executing contact sessions as scheduled. The session groups are spawned by the master group to provide low level control of ground station services during contact sessions. Their lifetime is for the duration of the contact session. The master and session message servers are externally accessible by station users and allow

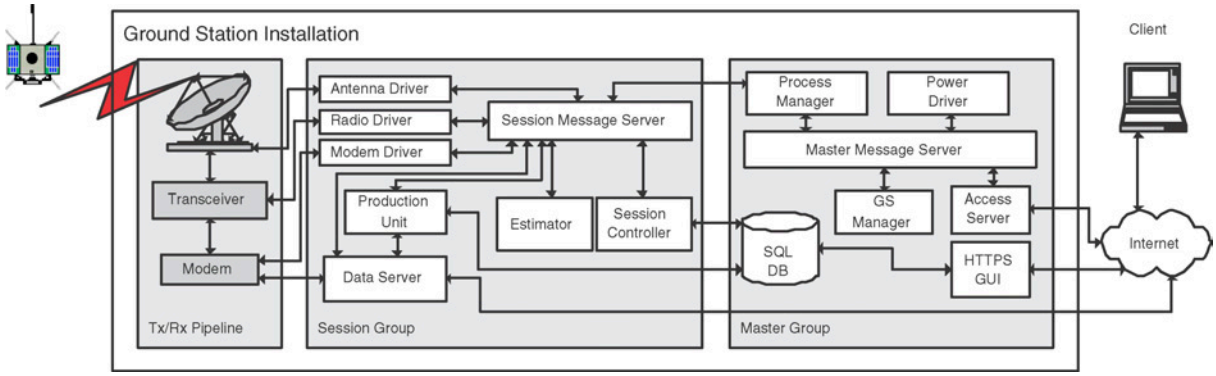


Fig. 3 High level block diagram of the Mercury ground station control system.

inter-layer control of services. The session server provides HL control and the master server provide SL control. The data server components provide external Internet access to data streams to and from the satellite over standard TCP connections.

Virtualization enforces strong inter-component isolation boundaries. Each of the back end Java components are run in their own separate Java Virtual Machine (JVM). This reduces the impact of a failed JVM (due to memory leaks or IO errors) to a single component. At a higher level, Mercury software components are spread across several VMware virtual machines[30] to isolate components from OS-level failures. An additional benefit of the encapsulation of Mercury in virtual machines is the level of indirection between the operating system and the hardware. The VMs can be run on any of the PCs within the ground station. Specially configured platforms, such as a web server system, are no longer confined to a particular CPU but can be migrated and backed up to other machines. This has decreased recovery time of failed CPUs. In the case of failure events such as failed power supplies, fans, hard drives, or motherboards, recovering from these requires simply running a backed-up copy of the VM on another CPU. The level of indirection helps us mitigate hardware failures that would affect single points of failure such as the MasterMessageServer.

B. High Availability Upgrades

A current Mercury development effort is the improvement of ground station availability, as it was not originally designed with high availability in mind. In this section, we discuss our application of recursive restarts (RR),[41] a recently proposed technique for achieving high availability that exploits partial restarts at various levels within complex software infrastructures to recover from transient failures and rejuvenate software components such that overall mean-time-to-recover (MTTR) is minimized.[6] We had two main goals in applying RR to Mercury. The first was to partially remove the human from the loop in ground station control by automating recovery from common transient failures that were restart curable. In particular, although all such failures are curable through a brute force reboot of the entire system, we sought a strategy with lower MTTR to minimize impact on operations. (More details will be given below). The second goal was to identify design guidelines and lessons for the systematic future application of RR to other systems.

1. Recursive Microreboots

It is common for bugs to cause a system to crash, deadlock, spin in an infinite loop, livelock, or to develop such severe state corruption (memory leaks, dangling pointers, damaged heap) that the only high-confidence way of continuing is to restart the process or reboot the system.[37,38,45,56] In fact, it is estimated that 60% of unplanned downtime in business environments is due to application failures and hardware faults, of which 80% are transient,[42,51,57] and therefore resolvable through reboot. Starting from this observation, we argue that in an appropriately designed system, we can improve overall system availability through a combination of reactively restarting failed components (revival) and prophylactically restarting functioning components (rejuvenation) to prevent state degradation that may lead to unscheduled downtime.[40] We define a framework that uses recursive restarts to recover a minimal subset of a system's components. And, if that doesn't help, we recursively recover progressively larger subsets.[6]

For a system to be recursively recoverable (RR), it must consist of fine grain components that are independently recoverable, such that part of the system can be repaired without touching the rest. This requires components to be loosely coupled and be prepared to be denied service from other components that may be in the process of restarting. We are encouraged by the increasing popularity of technologies that enable loosely coupled, modular systems, such as Sun J2EE[59] and Microsoft .NET.[50] A recursively rebootable system gracefully tolerates successive restarts at multiple levels, in the sense that it does not lose or corrupt data and does not cause other systems to crash. Due to its fine restart granularity, an RR system enables bounded restarts that recover a failed system faster than a full reboot.

2. Application of RR to Mercury

Recursive restarts are applicable to ground station systems because certain components have *windows of recovery*, where recovery within certain bounded windows of time result in failure free operation from the end users perspective. Thus, lowering MTTR so that it falls within these windows of recovery can directly affect user experience during an outage. For example, consider radio tuning systems where radios have bandwidths which describe the point at which signals are blocked by filters and no longer pass signals through to be processed. This bandwidth provides an error

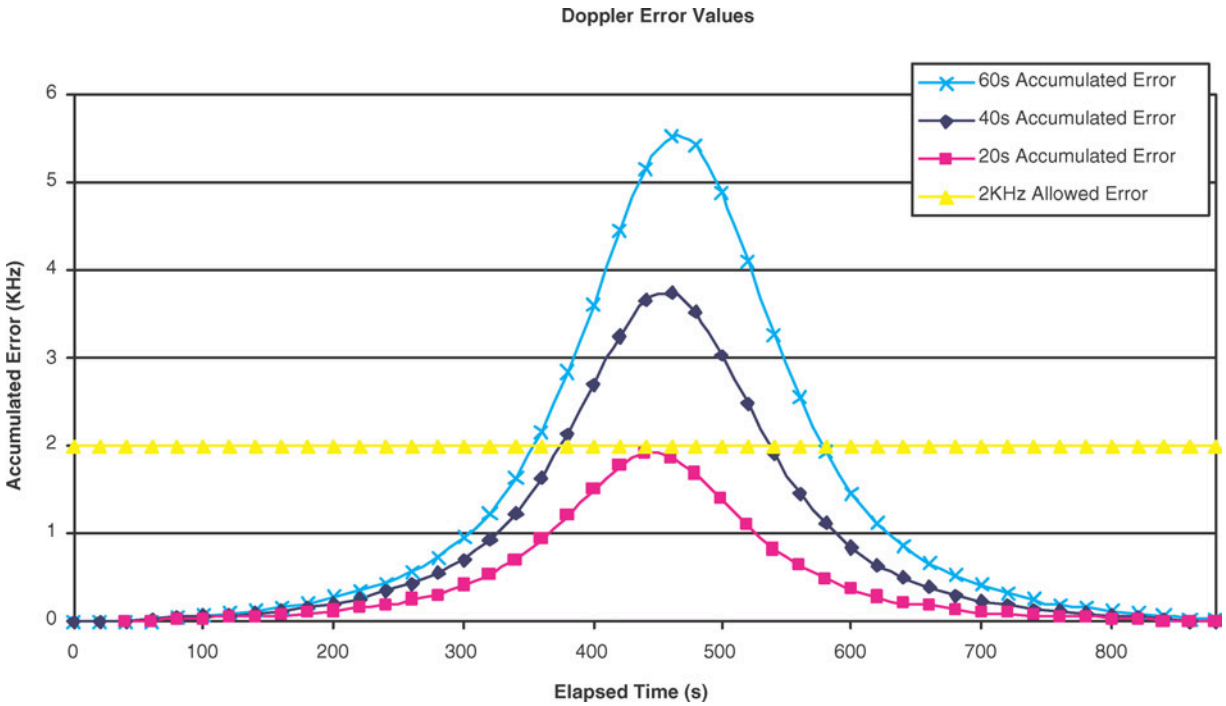


Fig. 4 Window of recovery for radio tuning.

margin for tuning radio systems where as long as the satellite transmissions are within the bandwidth, the signals are properly received. Radio tuning systems must account for the Doppler effect; the high velocity of the satellite shifts the frequency up when the satellite is approaching and down when the satellite is receding. Figure 4 shows a typical high elevation pass of LEO satellite where Doppler effects are most dramatic. The 2 KHz radio bandwidth margin is plotted along side 20s, 40s, and 60s error accumulations. This demonstrates a window of recovery for radio tuning to be approximately 20s. Unfortunately, not all components have these windows of recovery; any failure of them results in degradation of satellite contact sessions. Again, however, focusing on fast recovery of such systems will enable a reduction in data loss.

Due to our following the guidelines from section II.B, particularly *explicitly encapsulate state* and *preserve fault isolation boundaries*, Mercury was augmented with RR capabilities in a straightforward fashion. Important state variables used during ground station operations were placed in a transactional database thus enabling Mercury components to be restarted without loss of information. Virtual machine boundaries, both JVMs and VMware VMs, provide strong isolation boundaries that prevented memory leaks and other out of band errors from propagating. This helped contain errors and keep restart groups small. An independent component, the process manager, was developed to perform fault detection and recovery.

3. Restart Tree Transformations

A recursively restartable system can be described by a *restart tree*—a hierarchy of restartable components, in which nodes are highly fault-isolated and a restart at a node will restart the entire corresponding subtree (see table 1 for examples). A *restart cell* is the unit of recovery in a recursively restartable system which conceptually has a "button" that can be "pushed" to cause the restart of the entire subtree rooted at that node. Subtrees in the restart tree are called *restart groups*, in close analogy with process groups in UNIX where all component leaves are restarted as a unit.

Our application of RR to Mercury yielded several tree transformation techniques based on component MTTRs and MTTFs that reorganize restart groups of RR-amenable systems to improve overall system availability.[6] Our data is summarized in Tables 1 and 2. Depth augmentation of the restart tree, which results in the addition of new groups to

Table 1 Summary of restart tree transformations.

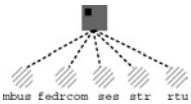
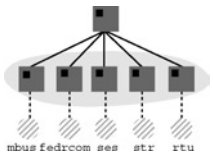
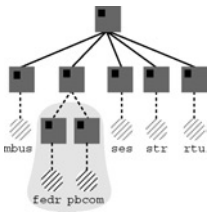
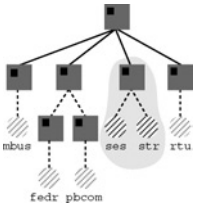
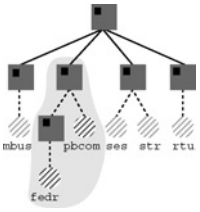
Original Tree	Augmentations		Reductions	
Tree I	Tree II	Tree III	Tree IV	Tree V
				
Original restart tree. Any component failure triggers a restart of the entire system.	Allows components to be independently restarted, without affecting others.	Saves the high cost of restarting pbcom whenever fedr fails (fedr fails often).	Reduces the delay in restarting component pairs with correlated failures (ise and istr).	Encodes information that prevents oracle from making guess-too-low mistakes.
Useful only if all component MTTRs are roughly equal.	Useful when $f_{A,B} > 0$ or $f_A + f_B > 0$, where f the probability of a component failure.	Useful when $f_{A,B} > 0$ or $f_A + f_B > 0$	Useful when $f_A + f_B \ll f_{A,B}$	Useful when oracle is faulty; i.e., it can guess wrong.

Table 2 Overall MTTRs (seconds). Rows show tree versions, columns represent component failures.

Tree	Oracle	mbus	ses	str	rtu	fedr	pbcom	fedrcom
I	Perfect	24.75	24.75	24.75	24.75	—	—	24.75
II	Perfect	5.73	9.50	9.76	5.59	—	—	20.93
III	Perfect	5.73	9.50	9.76	5.59	5.76	21.24	—
IV	Perfect	5.73	6.25	6.11	5.59	5.76	21.24	—
IV	Faulty	5.73	6.25	6.11	5.59	5.76	29.19	—
V	Faulty	5.73	6.25	6.11	5.59	5.76	21.63	—

the tree, provides reboot isolation of high MTTF components. Where possible, components that fail often are placed in separate restart groups to enable orthogonal restarts. Group consolidation merges restart groups with components that have failure dependencies. When failures of one component propagate to other components, the groups are merged to enable simultaneous reboot recovery thereby removing the detection time delay of dependent errors. Node promotion was developed to reduce the effects asymmetric MTTRs and MTTFs between related components and errors in diagnosis of which components to restart.

By employing recursive restartability we were able to improve recovery time of our ground station by a factor of four. Although we have not thoroughly measured the benefits resulting from automating the failure detection, we have observed them to be significant—in the past, relying on operators to notice failures added minutes or hours to the recovery time. One of the major failures was in the radio driver software that required a restart that initially took over 20 seconds to perform. As the example in Fig. 4 shows, this was beyond the window of recovery for radio tuning and resulted in a loss of contact with the satellite. Automating this recovery with the application of the restart transformations decreased the recovery time of common failures to less than 6 seconds—well within the window of recovery. Thus, the effects of the dominant Mercury failure mode were masked to the end user with sufficiently fast recovery.

4. Discussion

A promising feature of these techniques is that they required no modification to source code. Given the increasing trend toward complex, hard-to-manage software systems that integrate large numbers of COTS modules, we believe that recovery-oriented computing approaches hold an incredible amount of potential as a dependability technique in such systems. We draw some additional insights for applicability to other COTS-based systems.

Moving Boundaries—The most interesting principle we found was the benefit of drawing component boundaries based on MTTR and MTTF, rather than based solely on "traditional" modularity considerations such as state sharing. In transforming tree `III` to tree `IV`, we placed two independent components, `ses` and `str`, into the same recovery group. Presumably these two components are independent, but in practice a reboot on one component requires a reboot in the other for proper operation to resume. Therefore we partially "collapse" the fault-isolation boundary between them by imposing the new constraint that, when either is rebooted, the other one is rebooted as well. The effect was a decrease in recovery time from approximately 10 seconds to 6 seconds.

A dual of the above example is the splitting of the component `fedrcom` into the two separate components `fedr` and `pbcom`.[†] These two components are intimately coupled from a functionality perspective; either is useless without the other. That is why in the original implementation `fedrcom` was a single process; i.e., communication between the components that became `fedr` and `pbcom` took place by sharing variables in the same address space. Post-splitting, the two components explicitly communicate via IPC. This clearly adds communication overhead and complexity. The overhead was a minor increase in startup time, an additional 0.31 seconds due to an additional JVM, and an unmeasured but negligible increase in commanding latency due to the messages passing through another component. However, the split allows the two components to occupy different positions in the reboot tree, which in turn lowers MTTR. It is sufficiently lowered that the recovery time is within the window and all adverse effects are eliminated; thus well worth the trade-off.

We conclude that if a component exhibits asymmetric MTTR/MTTF characteristics among its logical subcomponents, rearchitecting along the MTTR/MTTF separation lines may often turn out to be the optimal engineering choice. Balancing MTTR/MTTF characteristics in every component is a step toward building a more robust and highly available system.

Not All Down Time Is the Same—Unplanned downtime is generally more expensive than planned downtime, and downtime under a heavy or critical workload is more expensive than downtime under a light or non-critical workload. In our system, downtime during satellite passes is very expensive because we may lose payload data or telemetry from the satellite. A large MTTF does not guarantee a failure-free pass, but a short MTTR increases the likelihood that we will not lose the whole pass as a result of a failure. As with other systems,[63] we have seen in this case study that from among systems with the same availability, those that have lower MTTR are often preferable because fast enough recovery eliminated adverse side affects of some failures.

C. Impact

Our GSML and Mercury development efforts have increased customization capabilities of ground station services, promoted development of COTS ground station solutions, and verified a mechanism for improving the availability of COTS-based systems without source code modifications.

1. Custom Composition

The abstraction of ground station communication channels into pipelines and the resulting network-centric implementation in Mercury increases flexibility in configuration of ground station data streams. Network enabled filters and software agents can now be implemented to augment and customize data streams. For example, during the Opal mission we experienced a communication anomaly that required us to physically rewire ground station components so that additional hardware could be inserted to help alleviate the problem. This manual process took several days

[†] Note, this was performed with a modification to a runtime configuration script, not an actual source code modification. Mercury driver code was designed to enable this type of splitting to provide flexibility for TCP or serial-based connections to hardware. This is used to run driver code on CPUs to which the hardware is not physically connected.

to implement on location at the ground station. With our Mercury-enabled station, we have performed this same procedure in software, external to the ground station, so that external hardware data manipulators are integrated into the data streams. In our next case study, we extend the concept of composition to create virtual ground stations spanning pipelines from physically separate ground station installations.

2. Enabling COTS Ground Station Solutions

Prior to this research effort, there were no concerted efforts in the university environment targeting ground station development. GSML and its reference implementation, Mercury, have now led to a prototype FGN, called the Mercury Ground Station Network (MGSN), that is supporting university satellite missions. We are partnering with the University of Würzburg in Germany on development efforts and working to install Mercury at ground stations operated by the Norwegian University of Science and Technology (NTNU) in Norway, with Aalborg University in Denmark, and with the University of Iowa.

Two development efforts of the Woyan project[22,31] underway at Würzburg demonstrate how GSML provides a structured method for developing commodity ground station services and for extending their capabilities. The first effort is developing centralized MGSN network services.[31] GSML provides the framework and Mercury provides a COTS solution to command and schedule ground station resources. Freed from these low-level implementation details, Würzburg researchers can focus on their core challenge, the development of optimal scheduling algorithms for multiple spacecraft supported by an FGN. A second Woyan development effort extends the layering of GSML beyond ground station systems to include spacecraft-specific capabilities.[22] Whereas the MGSN enables tasking of commodity ground station services, an additional higher layer developed as part of Woyan allows tasking of standardized spacecraft services. Currently, these services are mission agnostic and deal with transfer of data between ground and space systems. The system tasks MGSN stations to provide contact sessions where it can transfer data to and from satellites. It integrates a database with web services to provide long term storage and analysis for satellite data.

COTS development of GSML systems has been simplified through the use of an XML-based control language and the associated off the shelf tools. For example, over 60% of the custom Mercury code base (the back end Java software to control the station) is automatically generated from the XML Schema definition file of GSML. The 2000-line definition file is compiled with an open source databinding package[7] that creates a library of Java class files for manipulating GSML messages. This library is then wrapped with custom communication code that enables GSML messages to be transferred over TCP, UDP, SSL, HTTP, and SNMP. Ground station application developers are presented with an extensive library that enables GSML communication. Low-level implementation details of GSML message generation and delivering are masked by this automatically generated library thereby freeing developers from these issues. Any internal GSML structural changes are "instantly" compiled into source code and available to developers. This has facilitated the integration of non-Mercury systems such as Woyan into the MGSN.

Additionally, the breadth of XML tools has allowed us to run GSML applications on diverse computing platforms ranging from commodity PCs, to small embedded microcontrollers, to hand held PDAs. Most standard programming languages have XML libraries for parsing and generating XML messages. We currently use Java, PHP, and Perl for development of GSML applications. This facilitates development of additional GSML applications since it frees developers to use programming languages that best fit their needs.

3. Improved COTS Availability

Application of RR to Mercury completely removed the adverse effects of common software failures of a system that was not originally designed with high availability as a design criteria. Recovery time was reduced by a factor of four among some components to within the windows of recovery and all adverse effects of common ground station software failures were eliminated. By identifying mission-critical state and taking appropriate measures to protect this state during recursive recovery actions, our decomposition work enabled the application of RR. These enhancements to the Mercury system and improvements on availability were made without any source code modifications to the original Mercury system and our other COTS software. Because of this and the trend toward complex, hard-to-manage software systems that integrate large numbers of COTS components, we believe recovery-oriented computing approaches hold significant promise as a dependability technique for these COTS-based systems.

V. Case Study—End-to-End Mission

Our second case study is an end-to-end demonstration of our ground station technologies with QuakeSat-1, an extremely low-cost, experimental satellite that was launched in June of 2003 to study earthquake precursor signals in the Earth's magnetic field.[14,24] QuakeSat-1 is a first generation CubeSat,[26] weighs 5 kg, runs Linux on a COTS i486 CPU, and communicates over a half duplex, 9600 baud amateur radio communication channel. In this section, we describe the QuakeSat-1 integration into the Mercury system and the impact of our work on the success of this mission.

A. Mercury Mission Support

For QuakeSat-1 operations, the QuakeSat Operations Team (QOT) used two Mercury-enabled stations operated by Stanford University in Fairbanks, Alaska (FGS) and Stanford, California (SGS). The stations are typical COTS-based, OSCAR-class (Orbiting Satellite Carrying Amateur Radio) stations supporting satellite operations in the 2m and 70 cm (144–146 MHz and 435-438 MHz) amateur radio bands and providing 1200 and 9600 bps communication links cross or single band. One hundred percent of QuakeSat-1 contacts have been conducted with Mercury-enabled stations.

1. Legacy Software Support

Mercury components were used to network-enable legacy ground system components used by QOT. Their operations software communicated through serial ports and Linux kernel drivers to access the AX.25 networking capabilities of KISS[20] TNCs. Upgrading the drivers to support network-attached KISS TNCs would have required modifications of the kernel source code which was beyond the skill set of the QOT. Instead, a Mercury software component, the data server, was used to network-enable the software. The operations software communicated to the TNC through serial port 1. A serial cable was attached to serial port 1 and then to serial port 2, effectively looping data back into the computer. The data server was then connected to serial port 2 and to the ground station data server over the Internet. Thus data flowed from the operations software out serial port 1, into serial port 2, through the data server to the ground station, and vice versa. With a US$5 cable and a COTS Mercury component, QOT equipped their legacy mission operations software for network access without costly source code modifications.

2. Local Virtual Machine Execution

QOT provided a custom virtual machine with QuakeSat-specific operations software to the SGS for local VM execution.[‡] The operations software provides a GUI interface for human operators and is also capable of conducting autonomous QuakeSat contact sessions. The VM was used to provide local command and control of QuakeSat at the SGS during the initial vehicle checkout phase post launch. The QOT conducted these passes locally at the SGS to gain access to additional non-network ground station telemetry during this critical phase. The VM also provided autonomous local command and control of QuakeSat when human commanding was unnecessary or impossible (such as during network outages between SGS and the QOT operations center).

3. Not All Downtime is the Same

Soon after the deployment of the Fairbanks ground station (FGS), we discovered an aging effect in the operating system that resulted in complete contact session failures. Serial port communication would cease after several successful satellite contacts thus rendering the ground station useless and unable to perform contacts. If this happened during a pass, satellite contact would cease abruptly. Mercury component restarts failed to cure the failures and full system reboots were required. These took several minutes and resulted in significant data loss during passes. The failures were related to serial port accesses, which are critical for communicating with ground station hardware and were likely due to a faulty serial port driver. Internet connectivity was poor at best and made remote debugging very difficult.

Prophylactic reboots were deployed to mitigate the effects of these serial port failures. This type of software rejuvenation[47] has been shown to be useful in preventing systems prone to aging-related failures and was used here to mitigate the effects of OS-level aging that resulted in complete contact session failures. Mercury was tasked to

[‡] Unfortunately, FGS does not yet have virtual machine execution capabilities yet.

perform a full system reboot before conducting each contact session. This removed the aging effects in the operating system, and the ground station no longer experiences these serial port errors during contacts. Since the FGS runs on a single CPU system, the price for these reboots was temporary unavailability of the complete system. This only affected scheduling services of the ground station and overall improved the availability of the ground station during sessions.

4. Distributed VGS

With QuakeSat-1, we extended the concept of a virtual ground station to include pipelines from physically separate installations. The first step was post pass data stitching by QOT. QuakeSat-1 was commanded to dump data over FGS and SGS blindly. Mercury collected the data, archived it in a GSML data product, and made it available on-line. QOT downloaded this data and stitched together the downloaded files. The local archiving also mitigated effects of failures of external network connections by enabling longer term download options. The second step extended a Mercury component, the DataServer, to multiplex multiple transmit and receive pipelines in real time. This enabled seamless handoffs between ground stations and extended contact windows for QOT.

B. Impact

Mercury-enabled stations were a key enabler for the QuakeSat-1 mission by providing cost-effective, commodity ground station services that currently do not exist among commercial providers. Initial estimates for six months of ground station support by a commercial provider were approximately US$744K. At three fourths of QuakeSat-1's budget, this was well beyond the mission's grasp. The Mercury implementation of GSML provided additional commodity solutions for the QuakeSat mission. Direct cost comparison between the commercial provider and the MGSN services cannot be done since MGSN services were provided free to the QuakeSat-1 mission, but we do provide a brief qualitative comparison.

First, software integration costs between QOT and the Mercury stations were negligible. Remote connections to the station pipelines by legacy QOT software were enabled by Mercury components as described earlier. No new code was required and integration was preformed in several hours. Virtual machine installation was a straight forward file transfer into station servers. With several few hours of work, local application level support was provided. Though not addressed by this work, hardware integration efforts were negligible as well since both the stations and QuakeSat-1 use COTS amateur radio equipment for their pipelines. We also expect the cost per pass fees to be less. Mercury harnessed the idle resources of ground stations built for other missions and automated routine station tracking tasks. The cost to Stanford for providing support to QuakeSat-1 was negligible since it was all automated and performed remotely by QOT.

In addition to a commodity solution, the Mercury stations also provided enhanced pass support beyond existing commodity solutions. Local application support provided by VM execution mitigated wide area network failures and enabled local QuakeSat-1 control during mission critical check-out phases. Coordination between distributed ground station pipelines created greater contact windows and provided additional data download opportunities. QOT also used the layered services to customize ground station services. They used hardware level antenna commanding to reduce keyhole outages when the antenna reached the end of its range of motion and had to swing 360 degrees to recover.

Without the commodity services of the MGSN, the QuakeSat-1 mission would not have flown.[3] Existing commercial solutions were not cost effective in supporting the mission, and QuakeFinder lacked the funding and expertise to build their own ground station network. The MGSN harnessed the idle resources of global ground stations and enabled this high-risk experimental satellite to validate their mission concepts. The success of their mission has enabled QuakeFinder to obtain initial development funding for QuakeSat-2 which has an expected launch date of 2007.

VI. Related Work

Others have recognized these mission trends and ground station deficiencies and are working on possible solutions. There is a large movement to embrace terrestrial protocols that solve many of the networking challenges space

systems now face. The OMNI project at the NASA Goddard Space Flight Center is working on technology to operate satellite missions as nodes on the Internet. They have demonstrated end-to-end IP access on UO-36 and space shuttle flights.[10] The ChipSat mission operates with end-to-end IP networking as well.[19] This growing embrace of IP protocols complements our virtualization efforts.

Space-specific protocols have been extensively developed by CCSDS (Consultative Committee for Space Data Systems.[8]) They provide recommendations for all communication layers of space systems and a small set of space-specific COTS solutions are available. Specifically for ground stations, the Space Link Extension (SLE) has been developed to standardize transfer of CCSDS application data units between heterogeneous ground stations and end users.[28] SLE is limited to CCSDS systems and provides a narrow, limited interface to ground station capabilities. GSML provides SLE-like capabilities for a wider range of protocols and also extends ground station interfaces to lower and higher levels.

Network centric solutions for ground stations have been identified as a means of improving station configuration options and enabling new technology insertion.[2,5,17] This work assumes a network centric ground station and extends this concept to enable composition of ground station services across wide area networks. A "virtual software ground station" was developed that provides CORBA[61] interfaces to heterogeneous ground station hardware and software services.[1] GSML provides similar capabilities but is transport independent thereby enabling more flexibility in network communication architectures. GSML message can be transferred with CORBA and other networking protocols such as HTTP, FTP, and SMTP.

VII. Conclusions

The motivation for our work has been the reduction of communication as a constraint for space systems. We have targeted the lack of access to space communication and the decrease in system reliability resulting from increased system complexity. Our contributions are summarized below:

- Our decomposition of ground station services, informed by our extensive small satellite systems, and capturing of this in a portable, extensible framework was a key enabler in simplifying reuse of ground station services and improving portability across heterogeneous installations. This capability, combined with selective customization through virtual machine technology, allowed us to deliver a "just in time" ground station for QuakeSat-1 at a fraction of the price of current solutions. Integration costs were negligible due to GSML specifications and COTS hardware as compared to hundreds of thousands of dollars for commercial systems. Per pass costs were negligible due to automated pass execution and the harnessing of idle ground station resources.
- This decomposition is also informed by principles of robust system design, including explicit state encapsulation and strong inter-module fault isolation boundaries. Thus, our ground station reference implementation, Mercury, was a candidate for recursive recovery (RR), a high availability technique whose effectiveness in reducing recovery time has been demonstrated on research prototypes of Internet server systems. Augmenting Mercury to implement RR reduced recovery time of typical ground station software failures by a factor of four, dropping recovery time to within the "window of recovery" and effectively eliminating the adverse effects of these failures. Since the time of failures cannot be predicted, RR allowed us to mitigate the effects of the failures and greatly reduce their potential impact on ground station operations.

Our ground station architecture was able to harness the benefits of COTS, including rapid prototyping and deployment, while overcoming the challenges of COTS reliability and mission critical usage. Just as our CubeSat development work led to a drastic increase in university satellite research programs, our ground station research has laid a foundation for an increase in reliable, flexible access networks. Through the MGSN and projects like Woyan, commodity services are under development to expand access network capabilities. Our modularization work has enabled application of Internet high availability techniques to improve overall system performance. Doors have been opened to companies like QuakeFinder for supporting experimental satellite missions and to universities for graduate student research, and we expect in the near future that satellites will be as easy to access as websites are today.

Acknowledgments

Many thanks are extended to those who helped during our research. Prof. Bob Twiggs provided guidance on system design and use of SSDL ground station resources. Peder Linder drove the evolution of Mercury into an open-source system with the latest Internet technologies and laid the foundation for our extensive automatic code generation tools. The Stanford and Berkeley ROC groups provided valuable insights and partnerships, especially George Candea, Rushabh Doshi, Priyank Garg, and Rakesh Gowda. Thanks to Prof. Klaus Schilling and his students Andreas Lenz and Bartosz Wagner for their Mercury work and Woyan development. The QuakeFinder team played an important role in beta testing Mercury and stressing it under real world operations. Thanks for your patience. Thanks to NASA for support of this work through grant NAG3-2579 and an award from NASA's Graduate Student Researchers Program.

References

[1]Bernier, S., and Barbeau, M., A virtual ground station based on distributed components for satellite communications. In *Proceedings of 15th Annual AIAA/USU Conference on Small Satellites*, Logan, Utah, August 2001.

[2]Bhasin, K., and Hayden, J. L., Space Internet architectures and technologies for NASA enterprises. In *Proceedings of IEEE Aerospace Conference*, pages 931–941, Big Sky, Montana, 2001.

[3]Bleier, T., Quakesat-1 communication costs. Personal Communication, 2004.

[4]Brand, J., Kovarik, V., and Faist, J. Success oriented ground and space software defined architectures. Presented at the Ground Systems Architecture Workshop (GSAW), March 2004.

[5]Brill, M., NOCing the SOC. Presented at the Ground Systems Architecture Workshop (GSAW), February 2001.

[6]Candea, G., Cutler, J., and Fox, A., Improving availability with recursive micro-reboots: A soft-state system case study. In *Performance Evaluation Journal*, Summer 2003.

[7]The castor project. http://www.castor.org/, October 2004.

[8]Consultative Committee for Space Data Systems (CCSDS). http://www.ccsds.org.

[9]Cooning, M. G. C. R., Ground systems—providing effects to the warfighter. Presented at the Ground Systems Architecture Workshop (GSAW), Mar 2003.

[10]Criscuolo, E., Hogie, K., and Parise, R., Transport protocols and applications for Internet use in space. In *Proceedings of IEEE Aerospace Conference*, pages 951–962, Big Sky, Montana, 2001.

[11]Dnepr 2004 launch information. http://cubesat.calpoly.edu/_new/dnepr2004/index.html, November 2004.

[12]Cutler, J., Hutchins, G., Kitts, C., and Twiggs, R., Infrastructure for internet based operations. In *Proceedings of 14th Annual AIAA/USU Conference on Small Satellites*, Logan, Utah, September 2000.

[13]The dish: SRI international's antenna facility. http://www.essd.sri.com/dish/index.html.

[14]Flagg, S., Bleier, T., Cutler, J., and Dunson, C., Using nanosats (e.g. quakesat) as a proof of concept for space science missions. In *Proceedings of 18th Annual AIAA/USU Conference on Small Satellites*, Logan, Utah, August 2004.

[15]GNU radio—the GNU software radio. http://www.gnu.org/software/gnuradio/, November 2004.

[16]The ground station markup language (GSML). Draft at http://mercury.sourceforge.net/gsml/, October 2003.

[17]Hayden, J., Telecommunications for the CSOC, an internet interface to nasa satellites. In *Second Annual International Symposium on Advanced Radio Technologies*, September 1999.

[18]Information Sciences Institute. Internet protocol, Sep 1981. RFC 791.

[19]Janicik, J. W. J., The CHIPSat spacecraft design—significant science on a low budget. In *SPIE*, San Diego, California, August 2004.

[20]Karn, P., and Chepponis, M., The KISS TNC: A simple host-to-TNC communications protocol. In *The ARRL 6th Computer Networking Conference*, Redondo Beach, California, 1987.

[21]Kitts, C., Pranajaya, F., Townsend, J., and Twiggs, R., Emerald: An experimental mission in robust distributed space systems. In *Proc. AIAA Small Satellite Conference*, Logan, Utah, August 1999.

[22]Lenz, A., Database development for distributed satellite ground stations. Thesis, Julius-Maximilians University of Würzburg, January 2004.

[23]Linn, J., Privacy enhancement for internet electronic mail: Part i—message encipherment and authentication procedures, August 1989. RFC 1113.

[24]Long, M., Lorenz, A., Tapio, E., Jackson, K., Twiggs, R., and Bleier, T., A cubesat derived design for a unique academic research mission in earthquake signature detection. In *Proc. AIAA Small Satellite Conference*, Logan, Utah, August 2002.

[25]The mercury ground station control system. http://mercury.sourceforge.net/, October 2003.

[26]Nason, I., Puig-Sair, J., and Twiggs, R. J., Development of a family of picosatellite deployers based on the cubesat standard. In *Proceedings of IEEE Aerospace Conference*, Big Sky, Montana, 2002.

[27]Oltrogge, D. L., Spiral scan acquisition/tracking method for picosats. In *Aerospace Forum on Space Debris, Collision Avoidance, and Reentry Hazards*, November 2000.

[28]Space link extension executive summary, September 2001. CCSDS 910.0-Y-0.1, Draft Yellow Book.

[29]Swartwout, M., University-class satellites: From marginal utility to 'disruptive' research platforms. In *Proceedings of 18th Annual AIAA/USU Conference on Small Satellites*, Logan, Utah, August 2004.

[30]VMware. http://www.vmware.com.

[31]Wagner, B., Task planning and scheduling for satellite ground control stations. Thesis, Julius-Maximilians University of Würzburg, February 2004.

[32]Wertz, J. R., and Larson, W. J., editors. *Space Mission Analysis and Design*. Microcosm Press, El Segundo, California, 3rd ed., 1999.

[33]XML: the extensible markup language, 2003. http://www.w3.org/XML/.

[34]Zillig, D., and Benjamin, T., Advanced ground station architecture. In *Proceedings of SpaceOps*, Greenbelt, Maryland, November 1994.

[35]Adams, E., Optimizing preventative service of software products. *IBM Journal of Research and Development*, Vol. 28, No. 1, 1984, pp. 2–14.

[36]Adams, T., Igou, R., Silliman, R., Neela, A. M., and Rocco, E., Sustainable infrastructures: How IT services can address the realities of unplanned downtime. Research Brief 97843a, Gartner Research, May 15 2001. Strategy, Trends & Tactics Series.

[37]Blair, M., Obenski, S., and Bridickas, P., Patriot missile defense: Software problem led to system failure at Dhahran, Saudi Arabia. Technical Report of the U.S. General Accounting Office, GAO/IMTEC-92-26, GAO, 1992.

[38]Brewer, E., Lessons from giant-scale services. *IEEE Internet Computing*, 5(4): 46–55, July 2001.

[39]Brewer, E., Running Inktomi. Personal Communication, 2001.

[40]Candea, G., Cutler, J., Fox, A., Doshi, R., Garg, P., and Gowda, R., Reducing recovery time in a small recursively restartable system. In *Proc. International Conference on Dependable Systems and Networks*, pages 605–614, Washington, DC, June 2002.

[41]Candea, G., and Fox, A., Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *Proc. 8th Workshop on Hot Topics in Operating Systems*, pages 110–115, Elmau/Oberbayern, Germany, May 2001.

[42]Chou, T. C. K., Beyond fault tolerance. *IEEE Computer*, 30(4): 31–36, 1997.

[43]Cutler, J. W., Fox, A., and Bhasin, K., Applying the lessons of internet services to space systems. In *Proc. IEEE Aerospace Conference*, Big Sky, Montana, March 2001.

[44]Cutler, J. W., and Hutchins, G., Opal: Smaller, simpler, luckier. In *Proc. AIAA Small Satellite Conference*, Logan, Utah, September 2000.

[45]DiGiorgio, A., The smart ship is not enough. *Naval Institute Proceedings*, 124(6), June 1998.

[46]Gray, J., Why do computers stop and what can be done about it? In *Proc. Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986.

[47]Huang, Y., Kintala, C. M. R., Kolettis, N., and Fulton, D. N., Software rejuvenation: Analysis, module and applications. In *International Symposium on Fault-Tolerant Computing*, pages 381–390, Pasadena, CA, 1995.

[48]Internet Engineering Task Force. Hypertext transfer protocol (HTTP) 1.1, March 1997. RFC 2068.

[49]Lyu, M. R., editor. *Software Fault Tolerance*. John Wiley & Sons, New York, NY, 1995.

[50]Microsoft. *The Microsoft .NET Framework*. Microsoft Press, Redmond, WA, 2001.

[51]Milojicic, D., Messer, A., Shau, J., Fu, G., and Munoz, A., Increasing relevance of memory hardware errors. a case for recoverable programming models. In *ACM SIGOPS European Workshop "Beyond the PC: New Challenges for the Operating System"*, Kolding, Denmark, September 2000.

[52]Murphy, B., and Davies, N., System reliability and availability drivers of Tru64 UNIX. In *Proceedings of the 29th International Symposium on Fault-Tolerant Computing*, Madison, WI, February 1999. IEEE Computer Society. Tutorial.

[53]Murphy, B., and Gent, T., Measuring system and software reliability using an automated data collection process. *Quality and Reliability Engineering International*, 11: 341–353, 1995.

[54]Patterson, D., Brown, A., Broadwell, P., Candea, G., Chen, M., Cutler, J. Enriquez, P., Fox, A., Kcman, E., Merzbacher, M., Oppenheimer, D., Sastry, N., Tetzlaff, W., and Treuhaft, N., Recovery oriented computing (ROC): Motivation, definition, techniques, and case studies. Technical Report UCB/CSD-02-1175, UC Berkeley, Berkeley, CA, March 2002.

[55]Pradhan, D. K., *Fault-Tolerant Computer System Design*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1995.

[56]Reeves, G., What really happened on Mars? RISKS-19.49, January 1998.

[57]Scott, D., Making smart investments to reduce unplanned downtime. Tactical Guidelines Research Note TG-07-4033, Gartner Group, Stamford, CT, March 19 1999. PRISM for Enterprise Operations.

[58]Siewiorek, D. P., and Swarz, R. S., *Reliable Computer Systems: Design and Evaluation*. AK Peters, Ltd., Natick, MA, 3rd ed., 1998.

[59]Sun_Microsystems. J2EE platform specification. http://java.sun.com/j2ee/, 2002.

[60]Swartwout, M. A., and Twiggs, R. J., SAPPHIRE—Stanford's first amateur satellite. In *Proceedings of the 1998 AMSAT-NA Symposium*, Vicksberg, MI, October 1998.

[61]Vinoski, S., CORBA: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 14(2): 46–55, February 1997.

[62]W3C. XML Schema specification. http://www.w3.org/XML/Schema/.

[63]Xie, W., Sun, H., Cao, Y., and Trivedi, K. S., Modeling of online service availability perceived by web users. Technical report, Center for Advanced Computing and Communication (CACC), Duke University, 2002.