NASA/TM—2006-214083

Spread and SpreadRecorder
An Architecture for Data Distribution

Ted Wright
Glenn Research Center, Cleveland, Ohio

January 2006

The NASA STI Program Office . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA Scientific and Technical Information (STI) Program Office plays a key part in helping NASA maintain this important role.

The NASA STI Program Office is operated by Langley Research Center, the Lead Center for NASA's scientific and technical information. The NASA STI Program Office provides access to the NASA STI Database, the largest collection of aeronautical and space science STI in the world. The Program Office is also NASA's institutional mechanism for disseminating the results of its research and development activities. These results are published by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA's counterpart of peer-reviewed formal professional papers but has less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or cosponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services that complement the STI Program Office's diverse offerings include creating custom thesauri, building customized databases, organizing and publishing research results . . . even providing videos.

For more information about the NASA STI Program Office, see the following:

- Access the NASA STI Program Home Page at ***http://www.sti.nasa.gov***

- E-mail your question via the Internet to help@sti.nasa.gov

- Fax your question to the NASA Access Help Desk at 301–621–0134

- Telephone the NASA Access Help Desk at 301–621–0390

- Write to:
  NASA Access Help Desk
  NASA Center for AeroSpace Information
  7121 Standard Drive
  Hanover, MD 21076

NASA/TM—2006-214083

Spread and SpreadRecorder
An Architecture for Data Distribution

Ted Wright
Glenn Research Center, Cleveland, Ohio

National Aeronautics and
Space Administration

Glenn Research Center

January 2006

Available from

NASA Center for Aerospace Information
7121 Standard Drive
Hanover, MD 21076

National Technical Information Service
5285 Port Royal Road
Springfield, VA 22100

Available electronically at http://gltrs.grc.nasa.gov

Spread and SpreadRecorder
An Architecture for Data Distribution

Ted Wright
National Aeronautics and Space Administration
Glenn Research Center
Cleveland, Ohio 44135

Contents

1 Introduction

The Space Acceleration Measurement System (SAMS) project at the NASA Glenn Research Center (GRC) has been measuring the microgravity environment of the space shuttle, the International Space Station, MIR, sounding rockets, drop towers, and aircraft since 1991. The Principal Investigator Microgravity Services (PIMS) project at NASA GRC has been collecting, analyzing, reducing, and disseminating over 3 terabytes of collected SAMS and other microgravity sensor data to scientists so they can understand the disturbances that affect their microgravity science experiments. The years of experience with space flight data generation, telemetry, operations, analysis, and distribution give the SAMS/PIMS team a unique perspective on space data systems.

In 2005, the SAMS/PIMS team was asked to look into generalizing their data system and combining it with the nascent medical instrumentation data systems being proposed for ISS and beyond, specifically the Medical Computer Interface Adapter (MCIA) project. The SpreadRecorder software is a prototype system developed by SAMS/PIMS to explore ways of meeting the needs of both the medical and microgravity measurement communities. It is hoped that the system is general enough to be used for many other purposes.

2 The Task

The requirements for a space medical data system and a space microgravity measurement data system have much in common. They both must accept commands from authorized operators, collect data from sensors, save the data at least long enough for it to be transmitted to the ground, transmit the data to the ground, replay data transmissions that are not received, archive the data in an accessible format, and distribute the data to interested clients. The main differences are the sensors themselves (the data producers) and the clients that make use of the data (the data consumers).

2.1 High Level Design Goals

These features, and the desire to make a general purpose system that will be useful for things that are as yet unconceived, indicate that the system should be designed along the lines of the *end-to-end argument*[EndToEnd] (also known as the fundamental design principle of the Internet). That is, as much as possible, the distribution network should be a dumb system that just moves bytes without understanding them, and all the intelligence should be in the end points (the producers and consumers of data). Besides being the way to maximize network utility, the end-to-end design fits in well with the needs for data privacy for medical data. For maximum security, medical data should be encrypted at the source and decrypted at the destination. Since encrypted information looks like random bytes, any data system that depends on the details of this information will fail. The current SAMS/PIMS software is *not* designed in the end-to-end manner, and much of the complication in the PIMS data

processing is due to it having to extract and parse time stamp information from within the network packets for all the different types of sensors. Fixing this flaw should be a priority in any new design.

Another high level design requirement is that the system be flexible. The use cases indicate that astronauts would like to plug in medical sensors and have them be automatically configured. Likewise, space based displays of data should be able to be initiated or terminated at any time without affecting any other users of the system. Ground based medical personnel should be able to view ground based data without interfering with each other, and network and communication systems permitting, ground based medical personnel want to connect directly to flight based systems (in trauma situations). In summary, data sources and sinks can be expected to come and go at unexpected times, and the system should be designed so that this is not a problem. A good design for this type of system is a *publish/subscribe* architecture. Data sources *publish* data by connecting to some server when they have something to say, and data sinks *subscribe* to data "channels" on the server when there is data they wish to see. Example of publish/subscribe systems are the Internet's "Web Radio" and "Chat Rooms". In these cases, the radio "station" or room "name" are the "channel", and subscribers hear and see whatever is published on the channels they select. The current SAMS/PIMS software is only very approximately designed as a publish/subscribe system. In order to distribute the processing burden, it broadcasts data on the network and leaves it up to interested clients to receive and process the data, but there is no well defined interface for receiving only the data a client is interested in. A more well defined publish/subscribe data interface should be part of any new design.

Another requirement is that the system must collect data in space and send that data to the ground using the existing space communication architecture. It would be nice to have an unlimited bandwidth, full time, two-way, TCP/IP network connection between ground and space, but that is not the case. The nominal mode of operation for ISS based microgravity measurement systems is that they collect their data and hand it off to an ISS supplied Rack Interface Computer (RIC) for transmission to the ground. This is a one-way, asynchronous transmission. There is no acknowledgment back to the data system that the data was sent or successfully received. There can be no dependency on any particular timing of the data. In fact, there can be no assumption that a link is even possible at any given time. Depending on the orientation of the space station, there are different antennas available for communication at different times, and there are significant Loss Of Signal periods. Sending a command from the ground to a SAMS unit is even more of a problem. Commanding is normally disabled for security reasons. In order to send a command, a request must be submitted and a command window must be scheduled. Data can only be sent from ground to space during an authorized command window. It is true that in the case of a medical emergency all available bandwidth and commanding opportunities would be authorized and a direct TCP/IP link can be established (antenna orientations permitting), but no data system can assume that this will be its normal mode of operation.

2.2 Other Approaches

There are at least two other projects that have proposed solutions to the space medical data distribution problem. Glenn Research Center's *Project Rescue* uses a proprietary publish/-subscribe architecture called *Ring Buffered Network Bus* (RBNB)[RBNB] originally developed at NASA's Dryden Flight Research Center, now licensed by Creare, Inc. RBNB is a general purpose Java language data server with many features that go beyond the basics of a publish/subscribe architecture. Unfortunately, RBNB, at least as used by Project Rescue, is not designed in an end-to-end manner. The data server must know details about the data types and sizes of the data it is distributing. RBNB requires TCP/IP network connections.

Another proposed solution for medical data distribution is the *Switchboard* protocol developed by Stanford in collaboration with NASA's Ames Research Center and Johnson Space Center. Switchboard is a special purpose publish/subscribe system with predefined data types (not an end-to-end model). Switchboard is relatively immature, has few implemented features (no archiving, replication, time stamping, or data playback capability), and is written in the C# language (a proprietary Microsoft version of Java), limiting its usefulness in embedded systems. Switchboard requires TCP/IP network connections.

2.3 The SpreadRecorder Approach

The SAMS/PIMS team at GRC has developed another potential solution to this problem that uses a publish/subscribe architecture as well as an end-to-end model for data distribution. This approach uses an "off-the-shelf", free (open source) C language library called *Spread*[Spread] to provide the lower level publish/subscribe mechanism for data communication. Spread has bindings for several computer languages, so Spread clients are not limited to C.

Archiving, replication, time stamping and replay of data are provided by a program called *SpreadRecorder*, all without having to know the format of the data. Since the data system does not depend of the details of the data, it works equally well for microgravity measurements, medical data, and other data that was unanticipated when the system was implemented. In addition to the publish/subscribe data transfer mechanism, SpreadRecorder has an integrated RIC interface for downlinking data through the nominal one-way space to ground link available on the ISS, so it can move data from space to ground without a full bidirectional TCP/IP network connection.

The SpreadRecorder software developed by GRC consists of SpreadRecorder itself, several auxiliary programs (*ocaWatcher*, *ricWatcher*, *ricSimSim*) that simulate various aspects of the nominal ISS space-to-ground data system, and fourteen different Protocol Converters.

The SpreadRecorder system has been successfully tested with several data sources including a SAMS TSH-ES (Triaxial Sensor Head - Ethernet) microgravity measurement sensor, archived SAMS microgravity measurements replayed from a database, an OxyLink

Pulse Oxymeter sensor with a USB serial interface (which demonstrates the sensor being automatically configured and data acquisition happening in completely a "plug-and-play" manner), a Pulse Oxymeter connected through a remote Ethernet interface, archived blood pressure and ECG data captured from *Switchboard*, low frame rate video data from a Firewire connected DV camcorder, data collected via a wireless 802.11g connection from the Zin Technologies BioWatch sensor, and live data captured from the GRC developed PUMA1 and PUMA2 medical sensors over both USB and Bluetooth wireless interfaces.

SpreadRecorder has been demonstrated with several data sinks including remote networked graphical data displays showing realtime and playback gravity (acceleration) measurements, pulse displays, blood pressure and ECG charts, live video display, two different ISS RIC simulators to simulate the orbit to ground downlink, and graphical display of data on PocketPC Personal Digital Assistants linked to the system with a wireless Bluetooth interface.

SpreadRecorder has been installed and tested on disparate computer architectures including Dell personal computers with Intel microprocessors, Apple PowerBook and Mac Mini computers with PowerPC microprocessors, and the Project Rescue embedded system hardware (running on one of their Intel personality boards).

3 Spread

Spread is a messaging service for building distributed computer applications that are spread out across networks. It is free software developed by the Johns Hopkins University Center for Networking and Distributed Systems.

Spread implements a *publish/subscribe* architecture. Spread clients can send messages to a named channel (*publish*), and clients that have joined that channel (*subscribe*) will receive the messages. A single client can be both a producer and consumer of data, and multiple clients can use the same channel simultaneously.

From a programmer's perspective, Spread is extremely simple. It consists of only six functions:

connect to a Spread server

disconnect from a Spread server

join a Spread channel

leave a Spread channel

send a message to a Spread channel

receive messages from subscribed channels

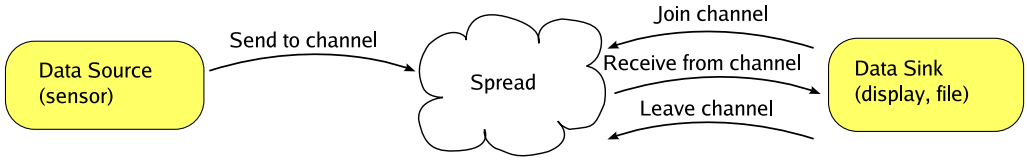

Figure 1: Spread message flow

The Spread Example Code Appendix shows how these functions are called from the Python Programming Language, which has a minimal amount of distracting superfluousity. The Spread message flow is illustrated in Figure 1 (which does not show *connect* or *disconnect*, because they are not Spread messages).

Data can be converted between their native input/output formats and Spread using *protocol converters*. A number of protocol converters (including ones for Ethernet, USB serial, RS232 Serial, and Bluetooth devices) are described in the Protocol Converters section.

Spread elegantly solves the problem of near real time communication between multiple software entities that do not need to be tightly coupled to each other. However, Spread is very simple, so there are many features that it does not provide. Spread does not provide any kind of data queuing for clients that are not joined to a channel. It does not provide any way of asking which channels are active. It does not provide archiving or playback of missed messages. These features are left to be provided by a Spread client program, such as SpreadRecorder.

4 SpreadRecorder

Archiving and playback of Spread message are accomplished using a Spread client program called SpreadRecorder. SpreadRecorder joins channels and records all messages sent to the channels in a database, along with a timestamp to keep track of when the message was received. It can be instructed to play back captured messages from the database into other Spread channels, so that spread clients can see messages that were sent when they were not listening. To keep the database tables from growing without limit, the oldest data in excess of a user specified number of messages is periodically purged from the database.

SpreadRecorder uses Spread messages for its user interface. It watches for commands on a Spread channel called *spreadRecorderCommand*, and sends its output to a Spread channel called *spreadRecorderStatus*. SpreadRecorder recognizes the following commands, which are fully described in the SpreadRecorder Commands Appendix:

help Shows the available commands and their options

shutdown Stops the SpreadRecorder program cleanly

add channel Adds a channel to the list of channels to record

remove channel Removes a channel from the list of channels being recorded

status Shows the list of channels being recorded and the list of playback threads

database status Shows the existing database tables and their statistics

playback Starts a new playback thread

stop thread Stops a playback thread

SpreadRecorder can play back data in two different formats, and has a provision for downlinking data though a high latency network such as the one on the International Space Station.

SpreadRecorder implementation details and design choices are described later. The SpreadRecorder Installation Appendix collects some notes about installing the software.

4.1 SpreadRecorder Data Formats

Spread can play back message in raw or cooked formats. A Spread client can join a channel that has raw data being played back by SpreadRecorder, and it can see the archived messages as if they were being generated now. However, there are some differences when compared to watching live data from a sensor. The data source will appear to be SpreadRecorder instead of the original sensor name, and the data will appear to be live, even though it was generated some time in the past. This is probably not what most data sinks would prefer.

A more useful format for playback data is the cooked format, which, in addition to the message, sends the original sending channel name, the original sender name, the data type, and the timestamp that the data first entered the system. Data sinks that work with cooked data can do more sophisticated things with it, such as generating graphs of the data with the correct time axis. For this reason, cooked format is the default for SpreadRecorder playback.

SpreadRecorder itself recognizes cooked data when it sees it on a channel (by using a particular data type for the Spread message), and treats the data differently when it is archived. When cooked data is recorded, the original channel name, sender, data type, and timestamp are preserved. Since SpreadRecorder can playback data to channels on other SpreadRecorders, cooked data playback can be used to replicate a SpreadRecorder database on a different system. This allows the system to be distributed across multiple computers and expanded as growth is required.

4.2 SpreadRecorder Downlink

SpreadRecorder has a feature that was added specifically for downlinking data from the International Space Station (ISS). The Spread protocol requires two-way communication between the clients and servers, making it inappropriate for high latency or essentially one way networks such as the ISS payload space-to-ground telemetry link. This means that except in unusual circumstances (requiring the right ISS network configuration and an authorized command window), the Spread protocol will not be used to send data from on orbit to the ground.

However, SpreadRecorder has an interface to the ISS EXPRESS Rack Interface Controller (RIC) that it can be used to playback data to the ground in much the same way as it would be played back to a Spread channel. If the name *downlink* is used as the destination channel name in a playback command, the messages being played will be formated for the RIC (split into small chunks, and prefixed with an appropriate header that includes enough information to reassemble the original message), and sent to a RIC connection for delivery to the ground. A provided program called RicWatcher running on the ground can reassemble the messages and inject them into a ground based Spread system. Non-emergency situations would use the RIC and have a ground based SpreadRecorder that mirrors the data from the space based SpreadRecorder. This is illustrated in Figure 2.

In emergency situations, where a two way space to ground link is permitted, ground based Spread clients could send commands or receive data directly from space based Spread data sources (or watch space based SpreadRecorder playback channels directly) since they will be essentially on the same local area network. In addition, there is a ground based program called OcaWatcher that can connect directly to the RIC interface on the space based SpreadRecorder to get any data sent to the downlink channel and can inject it into a different Spread system. This avoids going through the RIC and the UDP network data losses inherent in the RIC downlink data path.

5 Protocol Converters

Protocol converters are programs that connect the Spread system to devices or programs that do not natively use the Spread protocol. Several Python language protocol converters have been developed so far.

udpToSpread.py is a generic protocol handler that watches for UDP network data being sent on a particular network port, and inserts the captured UDP packets into Spread as Spread messages. The protocol handler is designed to be an importable module, so that one can make a customized version by importing the file and calling the udpToSpread function, passing in the port number to watch, the Spread server to connect to, and the name of the Spread channel that should receive the data. There is also

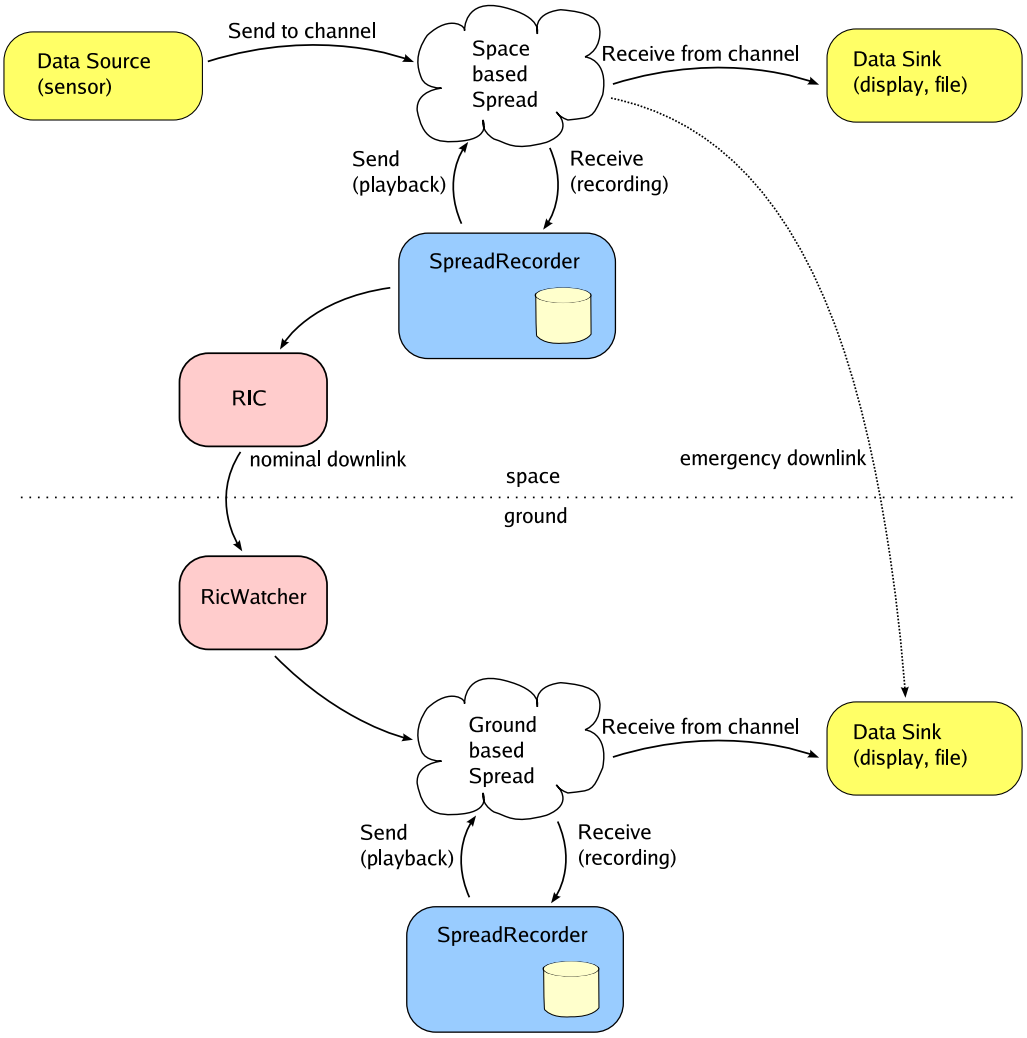

Figure 2: SpreadRecorder system data flow

an optional data transformation function that can be passed in if the data needs to be somehow manipulated before insertion into Spread.

spreadToUdp.py is a generic protocol handler that subscribes to a Spread channel, and broadcasts all the messages sent to that channel as UDP network packets (the opposite of udpToSpread.py). The protocol handler is designed to be an importable module, so that one can make a customized version by importing the file and calling the spreadToUDP function, passing in the port number on which to broadcast, the Spread server to connect to, and the name of the Spread channel that should be joined. There is also an optional data transformation function that can be passed in if the data needs to be somehow manipulated before being sent to the network.

tcpipToSpread.py is a generic protocol handler that watches for a TCP/IP network stream being sent on a particular network port, breaks the stream into chunks, and inserts the chunks into Spread as Spread messages. The protocol handler is designed to be an importable module, so that one can make a customized version by importing the file and calling the tcpipToSpread function, passing in the port number to watch, the Spread server to connect to, and the name of the Spread channel that should receive the data. There is also an optional data split function that can be passed in if the data needs to be broken in particular places (by default it is broken into fixed size chunks). Another option that can be passed in is the name of a TCP/IP server to contact in order to establish the connection. By default (no server passed in), the program becomes a server and waits for someone to establish a connection with it.

spreadToTcpip.py is a generic protocol handler that subscribes to a Spread channel, and sends all the messages sent to that channel as a TCP/IP network stream (the opposite of tcpipToSpread.py). The protocol handler is designed to be an importable module, so that one can make a customized version by importing the file and calling the spreadToTcpip function, passing in the port number on which to stream, the Spread server to connect to, and the name of the Spread channel that should be joined. There is also an optional data transformation function that can be passed in if the data needs to be somehow manipulated before being sent to the network. Another option that can be passed in is the name of a TCP/IP server to contact in order to establish the connection. By default (no server passed in), the program becomes a server and waits for someone to establish a connection with it.

pulseOxTcpipToSpread.py is a protocol handler designed to read data from a serial OxyLink Pulse Oxymeter with its RS232 interface plugged into a RS232 to Ethernet hardware converter. It imports and reuses tcpipToSpread.py .

pulseOxUsbToSpread.py is a full featured protocol handler designed to read data from a USB OxyLink Pulse Oxymeter. This protocol handler is designed to be started automatically by the Linux operating system's hotplug ("plug-and-play") subsystem,

so it can put itself into the background and run without user involvement ("daemonize" itself). The protocol handler is automatically run whenever the OxyLink device is inserted, and automatically stopped when the device is removed. It requires the installation of the PySerial[pySerial] library in order to read the USB serial port.

puma1serialToSpread.py is a protocol handler for collection of data from the PUMA 1 medical device, connected directly from the device's RS232 port to an RS232/USB converter. This protocol handler sends a few commands to the PUMA to turn on sensors and initiate data flow. It requires the installation of the PySerial library in order to read the USB serial port.

puma2serialBTToSpread.py is a protocol handler for collection of data from the PUMA 2 medical device, connected from the device's RS232 port to a Bluetooth wireless converter. Another Bluetooth wireless converter receives the data, converts it to RS232, and an RS232/USB converter converts it to USB. The PUMA devices do not use native Bluetooth networking, so they appear to Spread as USB serial devices. This protocol handler sends a few commands to the PUMA to turn on sensors and initiate data flow. puma2serialBTToSpread requires the installation of the PySerial library in order to read the USB serial port.

spreadSAMSsqlite.py is a protocol handler that reads archived SAMS II Microgravity Measurement data from a SQLite database and plays it into a Spread channel as if it were live measurements. It is used primarily for stress testing the system.

tshesTcpipToSpread.py is a protocol handler that reads live SAMS II TSHES Microgravity Measurement data from a TSHES sensor head. It imports and reuses tcpipToSpread.py, adding a custom data splitting function and the ability to daemonize itself.

videoSource.py is a protocol handler that reads images from a file (generated by a live DV camcorder at approximately 2 Hz), and inserts them into a Spread channel. The 2 Hz limit is due to the simplistic nature of this program. A smarter video source could supply full frame rate DV quality data, and the bandwidth should not be a problem for Spread.

videoDisplayWx.py is a protocol handler that watches a Spread channel with video image messages (such as one generated by videoSource.py), and displays those images on the computer screen. It requires the wxPython GUI library to be installed in order to show the images.

bluetoothSpreadServer.py is a bidirectional protocol handler that allows devices communicating with a wireless Bluetooth interface to participate in a Spread system. It advertises a spreadConnector service using native Bluetooth networking (not serial

port emulation), that allows connected devices to join and leave Spread channels and send and receive Spread messages. It uses the cooked data format to preserve things like Spread channel name and Spread sender during its data transfers. bluetoothSpreadServer.py requires the pyBlueZ[PyBlueZ] Python interface module for the BlueZ Bluetooth protocol stack, so it is unfortunately Linux specific.

spreadClient.py is a generic Graphical User Interface (GUI) to Spread systems (similar to the text based *spuser* program that comes with Spread). It is not exactly a protocol converter (unless *text* is considered a protocol), but it does recognize the cooked message format used by SpreadRecorder and displays the decoded original channel name, sender, time stamp and type extracted from cooked messages. It remembers joined channels and sent messages so that they only need to be entered once, and it comes pre-configured with SpreadRecorder's channels so that the user does not have to type long names like *spreadRecorderCommand*. The GUI is built using the PyGTK2 toolkit, so PyGTK2 and GTK2 must be installed in order to run the program.

6 Design Choices

This section documents some of the design choices and trade-offs made during the SpreadRecorder design and implementation.

6.1 Linux versus vxWorks

The real time operating system VxWorks was originally proposed as the target execution environment for the combined redesigned SAMS Control Unit and the Medical Computer Interface Adapter. However, a careful review of the MCIA requirements revealed no hard real time requirements. The current SAMS Interim Control Unit has been running for years on the (non-realtime) NetBSD Unix operating system, so it does not have any hard real time requirements.

Some thought was given to using a Real Time Operating System (RTOS) "just in case", but after contacting the GRC VxWorks software development experts, it was decided that that would incur too many penalties:

- Using a RTOS typically increases development cost and schedule by a factor of 3
- Commercial RTOS are expensive, VxWorks is typically 20–40 thousand dollars for one development system. Developers are more productive if they do not have to share some centralized system.
- Developing hard real time code is difficult. To avoid deadlocks and priority inversions, compromises are usually made to simplify the code (single thread of execution, one process, no memory protection, throwing away data that is not arriving fast

enough, etc.). These compromises increase the fatality of bugs and make adding future functionality harder.

- Ethernet is inherently non-deterministic, and cannot be done in a hard real time manner. Much of the code in this project depends on Ethernet and is inherently incompatible with hard real time constraints.

- Hard real time means that the system will die if a deadline is exceed. Soft Real Time means the system will continue in a degraded manner (maybe losing data) if a deadline is exceeded. For SAMS and MCIA purposes, losing data is preferable to aborting the program.

There is nothing about SpreadRecorder that is particularly hardware or operating system dependent. Some of the protocol converters are operating system dependent, but only because there is no common cross-platform way of implementing "plug-and-play" for things like USB devices. Where ever possible, cross-platform and cross-architecture languages and techniques were incorporated into SpreadRecorder. There are emerging standards for the "plug-and-play" configuration of network devices (such as *Zeroconf/Rendezvous*), and they should be considered for future development of data sources and sinks.

The Linux operating system was chosen for its reliability, cost, and wealth of software development tools. Red Hat's Fedora Core 4 distribution of Linux was chosen because of its support for both the Intel and PowerPC architectures, and because the PIMS developers were familiar with it from using it on the PIMS analysis cluster.

6.2 Python versus C (and others)

For SpreadRecorder to achieve maximum performance on a small embedded system, it should be written in the C programming language. C is so small and fast, it has been called "portable assembly language". However, it is not ideal for prototyping. Due to the constrained schedule and budget of this project ("months" and "none", respectively), a higher level language was required. It might be worthwhile to rewrite SpreadRecorder in C someday, but at this stage that would be a very premature optimization. Additionally, benchmarks may show that this system is I/O bound, which would mean there would be no speed benefit from such a rewrite.

The Python language is very high level, is portable, provides automatic memory management, and allows interactive experimenting with code. It is roughly an order of magnitude more dense than C, and has a correspondingly shorter development cycle. It is a strongly but dynamically typed language, so no development time needs to be spent specifying data types for every variable just to make the compiler happy. Python's two common high level data types (Lists and Dictionaries) make expressing many algorithms very simple. Python has been called "executable pseudo-code". For these rapid prototyping reasons, SpreadRecorder and all the protocol converters were written in Python.

Java provides automatic memory management and portability, but does not have most of the other benefits of Python. Java and Python are both byte compiled and run on virtual machines, so they have similar performance (significantly slower than C). Proprietary single operating system single architecture languages such as C# were not even considered.

6.3 Database versus a flat file system

The SAMS Control Unit currently stores all the measurements it receives in a circular buffer of fixed files. This is simple and fast. On the other hand, a home-brewed circular buffer does not lend itself to easy querying for data playback. When data is played back, someone will almost always want to limit the playback to a specific channel and time span. Databases are optimized for queries like that.

In addition, there can be more than one way a client may want to see the data. PIMS sometimes wants to see every recorded data item in the order it was received (for status messages). Sometimes PIMS wants to see only the most recently downloaded value for data with the same time stamp (to avoid duplicates with multiple data downlinks with overlapping time spans). Because databases make extracting only the desired data easier, it was decided that they were worth the added complexity.

6.4 MySQL versus SQLite

Given that a database is going to be used by SpreadRecorder, a particular implementation must be chosen. PIMS uses the MySQL[MySQL] database for their data collection. Besides being fast and reliable, there was already existing PIMS code that worked around MySQL's known quirks. However, MySQL is almost too heavyweight for a small embedded system. It requires some initial configuration and a server process that is always running in the background.

A C library based database such as SQLite[SQLite] would be much more lightweight and suitable for small embedded systems. This system requires no setup or other processes running, and it stores its data in regular files. SpreadRecorder does not use any complicated SQL procedures, so SQLite's features are adequate. Performance is, in some cases, even better than MySQL. SQLite looked like an obvious choice.

Despite concentrating all database access in one function to ease interfacing and using the supposedly standard SQL language for queries, it turns out that switching from one database to another is not very transparent. Databases apparently make a lot of underlying assumptions about how they are going to be used. An early version of SpreadRecorder was written using SQLite, but it could not be made to run reliably for periods of more than a few days. After wasting a few weeks trying to eliminate all the bugs, development of SpreadRecorder was switched back to MySQL, which was heavy, but ran with SpreadRecorder for months at a time processing billions of records without a fault. When the time comes to optimize the system, this issue should be revisited.

Appendix A Spread Example Code

Spread is implemented as a portable library written in the C programming language. It has bindings[Spread bindings] available that allow it to be called from the Python programming language. Python will be used for code examples, and is shown inside boxes for clarity. Other languages have a similar interface, but probably have more stuff obscuring the Spread calls.

The Spread module must be imported to make the Spread functions available.

```
from spread import *
```

A.1 Connect To A Spread Server

By default, the *connect* function connects to a Spread server running on 'localhost'. You can pass 'port@host' to the function to get a connection to a different server. The variable *mbox* is just a name that will be used to refer to the connection.

```
mbox = connect()
print 'connected as:', mbox.private_group
```

A.2 Disconnect From A Spread Server

The *disconnect* function is the opposite of *connect*. It cleanly shuts down the connection to Spread.

```
mbox.disconnect()
```

A.3 Join A Spread Channel

If you *join* a channel, you are telling Spread that you want to receive messages sent to that channel. After joining, you must call *receive* often enough so that Spread does not queue up too many outstanding messages for you (1000 is the default), or else Spread will disconnect you. The name of the channel you wish to join is passed to this function.

```
mbox.join('spreadtest')
```

A.4 Leave A Spread Channel

The *leave* function is the opposite of *join*. It tells Spread not to send you any more messages for that channel. The name of the channel you wish to leave is passed to this function.

```
mbox.leave('spreadtest')
```

A.5 Send A Message To A Spread Channel

Sending a message to a channel is done using the *multicast* function. You do not have to join a channel in order to send messages to it. Passed into this function are: the message type (a constant indicating 'safe message', which is the only type of Spread message used by SpreadRecorder), the channel name, and the message itself.

```
mbox.multicast(SAFE_MESS, 'spreadtest', 'Hello World')
```

A.6 Receiving Message From A Joined Spread Channel

Theoretically, a message is received by simply calling the *receive* function.

```
m=mbox.receive()
```

However, there are complications. The *receive* function is a *blocking* call, which means if you call it when there are no messages to receive, your thread of execution will be suspended until a message arrives. Unless that behavior is what you want, it is best to only call *receive* when a message has already arrived.

Spread provides a *poll* function for checking if a message has arrived, but the documentation suggests that it not be used because it gives no indication that messages are not ready because the network connection was terminated. A better way to check for messages is to use the standard Unix *select* function, which has a Python implementation even on non-Unix systems. The *select* function will throw an exception if the network connection is unexpectedly terminated.

The *receive* function blocks if it is called too often, but it has another problem if it is not called often enough. Spread will only queue up a certain number of messages for a client before it decides that the client has died and the connection to the client should be destroyed. If a client program subscribes to a Spread channel and then does not call *receive* often enough, it will be disconnected. For this reason, when processing Spread messages, it is a good idea to keep calling *receive* and handling the messages in a loop for as long as messages are available. When there are no messages left to process, then the program can go on to other duties.

The last complication is that there are actually two types of Spread messages that can be returned by the call to *receive*. In addition to the *regular* messages that a client is expecting, there will also be *membership* messages that will appear when clients join and leave subscribed channels. Most clients will want to ignore these membership messages, which can be recognized by checking the message type.

In a reality, there is more to receiving than just calling *receive*.

```
import select
# timeout=0 in select is equivalent to poll
r, w, x = select.select([mbox.fileno()],[],[],0)
while r:
    m=mbox.receive()
    # that is all as far as receiving is concerned
    # now display what was received:
    if isinstance(m, RegularMsgType):
        print 'Regular Message sender:', m.sender
        print '  groups:', m.groups
        print '  type:', m.msg_type
        print '  length:', len(m.message)
        print '  message:', m.message
    elif isinstance(m, MembershipMsgType):
        print 'Membership Message reason:', m.reason
        print '  group:', m.group
        print '  members:', m.members
        print '  extra:', m.extra
    # check if more message are ready
    r, w, x = select.select([mbox.fileno()],[],[],0)
```

Appendix B SpreadRecorder Commands

SpreadRecorder uses Spread messages for its user interface. It watches for commands on a Spread channel called *spreadRecorderCommand*, and sends its output to a Spread channel called *spreadRecorderStatus*. Any Spread client can send a command to the *spreadRecorderCommand* channel, and any client can join the *spreadRecorderStatus* channel to see messages from SpreadRecorder.

Commands are text based so that they can be easily remembered and typed.

B.1 Help Command

Help simply lists all of the command recognized by SpreadRecorder, along with the default values of any options the command will recognize.

There are no optional parameters for the help command.

```
help
```

B.2 Shutdown Command

The shutdown command cleanly stops the SpreadRecorder program. Clients should normally avoid doing this.

The list of channels that are being recorded is saved when SpreadRecorder shuts down, and automatically restored when it starts up again. However, the list of active playback threads is not saved or restored.

There are no optional parameters for the shutdown command.

```
shutdown
```

B.3 Add Channel Command

The add channel command adds the given channel name to the list of channels that SpreadRecorder will archive. SpreadRecorder will create a database table to hold the messages for the added channel (if it does not already exist), join the Spread channel, and write every message it receives from that channel to the table.

In order to keep the tables from growing until disk space is exhausted, the oldest messages in excess of a table specific limit are periodically deleted from the table. The default limit is 50000 messages, but may be changed when the channel is added by supplying a limit option.

The name of the channel to add is a required parameter for the add channel command. An optional parameter is *limit*.

```
add channel channelName [limit=100000]
```

B.4 Remove Channel Command

The remove channel command removes the given channel name from the list of channels that SpreadRecorder will archive. SpreadRecorder will leave the spread channel. The database table associated with the channel is not removed.

The name of the channel to remove is a required parameter for the remove channel command.

```
remove channel channelName
```

B.5 Status Command

The status command displays the current status of SpreadRecorder. It shows the number of messages that it has seen but not yet recorded, the total number of messages recorded, and the number of messages waiting for pickup in the downlink queue.

The next section of the status output shows a list of all the channels it is currently watching, along with the database size limit for each channel.

The last section shows a list of all currently running threads. Each playback thread is listed with a name, and a list of the parameters that were given to start the playback (to help tell them apart). The name is used by the stop thread command.

The status command has no required parameters. It can optionally be given a channel name to send its results to instead of sending them to the default status output channel (spreadRecorderStatus). This option is intended to help programs that poll for status to keep their information separate.

```
status [optionalOutputChannelName]
```

Here is an example of the status command's output as it would appear in the *spreadRecorderStatus* channel:

```
status command received from #r2331-12#wanda
  messageQueue has 0 unprocessed messages, 312502 processed
       ricQueue has 0 unprocessed messages
  recording 8 channels:
    channel ECG (database limit 50000)
    channel PulseOxUSB (database limit 50000)
    channel SAMS_II_121f02 (database limit 100000)
    channel SAMS_II_121f05 (database limit 50000)
    channel counter (database limit 50000)
    channel spreadRecorderCommand (database limit 50000)
    channel spreadRecorderStatus (database limit 50000)
    channel tshes01 (database limit 50000)
  active threads ['SpreadRecorder', 'Thread-1', 'DownlinkConnector', 'ProcessQueue',
                  'DatabaseReaper']
    threadName Thread-1: playback PulseOxUSB POpb rate=1 formatted=0 keepGoing=loop
```

B.6 Database Status Command

The database status command displays the current status of the database tables being used by SpreadRecorder. This command may take some time to execute, because it examines and collects statistics for all the tables in the SpreadRecorder database that match SpreadRecorder's table layout (even if the channel is not currently being recorded).

For each table, database status shows the table name, the time stamps of the oldest and newest messages in the table, and the number of messages in the table.

The time stamps are represented as floating point numbers that are "Unix time" (the number of seconds since 1970) plus the fraction of a second represented as digits after the decimal point. The SpreadRecorder database format also has a serial number field that is used to keep messages ordered even if they have the same time stamps, but this is transparent to users.

The database status command has no required parameters. It can optionally be given a channel name to send its results to instead of sending them to the default database status output channel (spreadRecorderStatus). This is intended to help programs that poll for status to keep their information separate.

```
database status [optionalOutputChannelName]
```

Here is an example of the database status command's output as it would appear in the *spreadRecorderStatus* channel:

```
database status command received from #r2331−12#wanda
  ECG 1120217539.31 1120664175.98 7836
  PulseOxUSB 1125580368.45 1129829599.47 2600
  SAMS_II_121f02 1121447707.46 1125662553.25 100000
  SAMS_II_121f05 1120217539.86 1120664175.12 504
  SoundData None None 0
  counter 1124202099.84 1124203102.77 50000
  spreadRecorderCommand 1120664842.88 1131044442.51 1638
  spreadRecorderStatus 1120664141.19 1131044442.51 1836
  tshes01 1120217539.34 1120664175.94 571
```

B.7 Playback Command

The playback command reads archived messages from the database, and sends them to a given Spread channel (or as a special case, to the downlink queue). There are many optional parameters to change the default playback behavior.

The name of the source channel to play back is a required parameter for the playback command, as well as the name of the destination channel where the messages will be sent. The destination channel can optionally have a network port and host name appended to it, separated by '@' characters, for playback to a separate Spread system running on another computer.

Optional parameters *startTime* and *stopTime* can be used to limit the range of data to be played back. Time is specified as "Unix time": the number of seconds since January 1, 1970. You can use *now* as a shortcut for the current time.

The optional *rate* parameters sets the number of messages per second to play back. The default is 50. It is unrelated to the rate that the messages arrived in the system.

The optional *formatted* parameter determine whether to play back raw messages (formatted=0), or cooked messages (formatted=1). Cooked messages are the default.

The optional *keepGoing* parameter determines what will happen when all the data has been played back. If left at its default value of 0, the playback thread will terminate. If it is set to 1 (*keepGoing=1*), the thread will stay alive and wait for more data to arrive, which will be played back immediately. If it is set to loop (*keepGoing=loop*), it will start over at the beginning of the database table and repeat (this is useful mostly for testing purposes).

Another optional parameter is *reconnect*, which will try to reestablish a connection to a remote destination channel if the connection is lost (this defaults to 1, which means it will try to reconnect).

The optional *mergeTimestamps* parameter determines what will happen if multiple messages have the same time stamp. When left at the default value of 0, all messages with the same timestamp will be played back in the order they were received. If *mergeTimestamps* is set to 1, only the most recently received message with a particular time stamp will be played. This is useful if you want more recent data to replace older data, rather than add to it.

```
playback  sourceChannelName destinationChannelName[@Port@Host]
  [reconnect=1]  [rate=50]  [formatted=1]  [mergeTimestamps=0]
  [keepGoing=0] [stopTime=9999999999.0] [startTime=0.0]
```

B.8 Stop Thread Command

The stop thread command stops a thread that was started with the playback command.

When the playback command is issued, it starts a separate thread of execution to do the playback, and it assigns a name to the thread. Depending on the options issued to the playback command, the thread may not terminate on its own. These threads can only be terminated by calling the stop thread command with their name. Names of threads may be found using the status command.

The name of the thread to stop is a required parameter for the stop thread command. Playback thread names are automatically generated, and will normally look something like *Thread-22*.

```
stop  thread  threadName
```

Appendix C SpreadRecorder Installation

Some of these installation instructions are specific to the environment used at the NASA GRC, but similar procedures should work elsewhere. The GRC Spread installation uses the Fedora Core 4 Linux operating system running on computers with Intel and PowerPC microprocessors.

C.1 Spread Installation

Both Spread and the Spread binding for Python can be installed on Linux using Red Hat Package Manager (RPM) files. Install the binary RPMs appropriate for your architecture with the *rpm -Uvh ...* command. These commands must be issued from a root shell.

The RPM files are available from:
`http://parrot.grc.nasa.gov/linux/fedoraCore-4/updates/local` or
`http://parrot.grc.nasa.gov/linux/fedoraCore-4/ppc/updates/local`

For the Fedora Core 4 Intel architecture you need:

- spread-3.17.3-3.i386.rpm
- spread-devel-3.17.3-3.i386.rpm
- SpreadModule-1.5-1.i386.rpm

For the Fedora Core 4 PowerPC architecture you need:

- spread-3.17.3-3.ppc.rpm
- spread-devel-3.17.3-3.ppc.rpm
- SpreadModule-1.5-1.ppc.rpm

Source RPMs can be used to generate binary RPMs for other distributions and architectures (rebuild them with the *rpm –rebuild ...* command):

- spread-3.17.3-3.src.rpm
- SpreadModule-1.5-1.src.rpm

After installing the Spread RPMs, the */etc/spread.conf* file must be edited to include the IP address and network port of the Spread server. The localhost entry must be commented out. It should look like this:

```
#Spread_Segment  127.0.0.255:4803  {
#           localhost   127.0.0.1
#}

Spread_Segment  139.88.79.255:4803  {
          wanda       139.88.79.81
}
```

Edit the lines in the SpreadRecorder *settings.py* file to reflect your Spread server name and server port (as specified in */etc/spread.conf*), and MySQL host.

Next, the Spread server must be turned on:

```
service  spread  start
chkconfig  spread  on
```

SpreadRecorder uses an SQL database. In theory, switching to different databases should be easy. Database access is all done through one function, which submits "standard" SQL commands to the database and returns the results. In practice, the SQL required for different databases is not very standard, and different databases make different assumptions about how things like transactions are handled. SpreadRecorder currently works with the MySQL database, which should be set up on the same computer running SpreadRecorder for best performance.

Whatever database is used will require bindings to the Python language. SpreadRecorder currently uses MySQLdb, which must be installed. MySQLdb comes with Fedora Core 4 Linux, but is not installed by default. If your computer is configured to access the Internet (including any needed HTTP proxy environment variables), then MySQLdb can be installed by doing:

```
yum  install  MySQL-python
```

MySQL needs to be turned on and the empty SpreadRecorder database must be created:

```
service  mysqld  start
chkconfig  mysqld on
mysql -e "create  database  spreadRecorder"
```

Reading serial and USB ports requires setting things like baud rates to match the sensor device. Protocol handlers written in Python can make use of the pySerial module to make that easy. This is required for the pulseOxTcpipToSpread.py protocol handler.

PySerial is installed using the standard Python distutils method. Download the source file pyserial-2.2.zip, then:

```
unzip  pyserial -2.2.zip
cd  pyserial -2.2
python  setup.py  install
```

The Spread system should now be ready to go. System configuration is finished, so you can now leave the root shell and work with Spread using a regular user ID.

You can use either the GUI based *spreadClient* program or the text based *spuser* program to send Spread messages and join Spread channels. Join the *spreadRecorderStatus* channel to see messages from SpreadRecorder:

```
spuser −r −s 4803@spread.server.com
```

SpreadRecorder is not yet written as a daemon, so just start it in a terminal window:

```
./ spreadRecorderMysql.py
```

The Unix *screen* utility is useful for running programs in terminal windows like this while giving you the ability to disconnect from the computer and resume later (without stopping the running program).

C.2 PulseOxUSB Setup

For the OxyLink Pulse Oxymeter to operate as a plug-and-play device on Linux, edit the file */etc/hotplug/usb.usermap* and add a line that says:

```
pulseOxUSB 0x0003 0x0403 0x6001 0x0000 0x0000 0x00 0x00 0x00\
   0x00 0x00 0x00 0x00000000
```

This will cause the *pulseOxUSB* script to be called whenever the device is inserted or removed. Copy SpreadRecorder's *pulseOxUSB* script to the */etc/hotplug/usb* directory so that it will be found, and edit it to make sure it references the correct location of the *pulseOxUsbToSpread.py* file.

The *pulseOxUsbToSpread.py* is a full featured Unix daemon program, that will start and put itself in the background so that it can run without anyone being logged in. All of the protocol handlers (and SpreadRecorder itself) should eventually be modified this way, given more development time.

Data should start flowing on the *PulseOxUSB* Spread channel a few seconds after the device is plugged in. Be sure to send a message (one time) to SpreadRecorder on the *spreadRecorderCommand* channel telling it to record the *PulseOxUSB* Spread channel. This can be done from the command line:

```
./ sendSrCommand.py 4803@spread.server.com add channel PulseOxUSB
```

C.3 TSHES Setup

The SAMS TSH-ES acceleration sensor must be configured with an IP address (unless you want to accept its default). The procedure for doing this is:

1. Plug in the power for the TSH. It will start up with an IP address of 10.11.240.1XX, where XX is the serial number of the TSH.

2. Temporarily configure a computer so that it is on the 10.11.240/24 subnet (e.g., with an IP address of 10.11.240.100). Plug the computer into a hub with the TSH.

3. From the temporary computer, telnet to 10.11.240.1XX, and log in (user=root, password=root)

4. Run the command *ifconfig eth0 139.88.79.242* to give the TSH the new IP address (the new address is 139.88.79.242, in this example).

5. The TSH is now waiting for connections on the new address (and the telnet connection will appear to be hung). Without powering-off the TSH, you can move its network cable to a different hub, if necessary.

Edit the last line of *tshesTcpipToSpread.py* to reflect the IP address and port number of the TSH, or else pass the IP address on the command line.

The *tshesTcpipToSpread.py* protocol handler can be run as a daemon, but it is not started automatically. To start it as a daemon, run:

```
./tshesTcpipToSpread.py daemonize=start
```

Alternatively, just start it running in a terminal window:

```
./tshesTcpipToSpread.py
```

Data should start flowing on the *es01* Spread channel. Be sure to send a message (one time) to SpreadRecorder on the *spreadRecorderCommand* channel telling it to record the *es01* Spread channel. This can be done from the command line:

```
./sendSrCommand.py 4803@spread.server.com add channel es01
```

References

[EndToEnd] *End-to-end arguments in system design.* Jerome H. Saltzer, David P. Reed, and David D. Clark. ACM Transactions on Computer Systems 2, 4 (November 1984) pages 277-288. An earlier version appeared in the Second International Conference on Distributed Computing Systems (April, 1981) pages 509-512.
Online at: `http://web.mit.edu/Saltzer/www/publications/endtoend/endtoend.pdf`

[RBNB] *Ring Buffered Network Bus*. Lawrence C. Freudinger (Dryden Flight Research Center) and Matthew J. Miller, Ian A. Brown, and William R. Baschnagel (Creare, Inc.)
Online at: urlhttp://outlet.creare.com/rbnb/WP/WebWP/rbnbwp.html

[Spread] *The Spread Toolkit*. Center for Networking and Distributed Systems (CNDS) at Johns Hopkins University.
Online at `http://www.spread.org`

[Spread bindings] *Spread Toolkit bindings for the Python Programming Language*.
Online at `http://www.zope.org/Members/tim_one/spread`

[MySQL] *MySQL Database*.
Online at `http://www.mysql.com`

[SQLite] *SQLite Database*.
Online at `http://www.sqlite.org`

[MySQL bindings] *MySQL Database bindings for the Python Programming Language*.
Online at `http://sourceforge.net/projects/mysql-python`

[pySerial] *pySerial serial port support for the Python Programming Language*.
Online at `http://pyserial.sourceforge.net`

[PyBlueZ] *BlueZ Linux Bluetooth support for the Python Programming Language*.
Online at `http://org.csail.mit.edu/pybluez`

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (*Leave blank*)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	January 2006	Technical Memorandum

4. TITLE AND SUBTITLE

Spread and SpreadRecorder
An Architecture for Data Distribution

5. FUNDING NUMBERS

WBS 825080.08.02

6. AUTHOR(S)

Ted Wright

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
John H. Glenn Research Center at Lewis Field
Cleveland, Ohio 44135–3191

8. PERFORMING ORGANIZATION REPORT NUMBER

E–15420

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
Washington, DC 20546–0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

NASA TM—2006-214083

11. SUPPLEMENTARY NOTES

Responsible person, Ted Wright, email: Theodore.W.Wright@nasa.gov, organization code PTH, 216–433–5341.

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified - Unlimited
Subject Category: 62

Available electronically at http://gltrs.grc.nasa.gov

This publication is available from the NASA Center for AeroSpace Information, 301–621–0390.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 words)***

The Space Acceleration Measurement System (SAMS) project at the NASA Glenn Research Center (GRC) has been measuring the microgravity environment of the space shuttle, the International Space Station, MIR, sounding rockets, drop towers, and aircraft since 1991. The Principle Investigator Microgravity Services (PIMS) project at NASA GRC has been collecting, analyzing, reducing, and disseminating over 3 terabytes of collected SAMS and other microgravity sensor data to scientists so they can understand the disturbances that affect their microgravity science experiments. The years of experience with space flight data generation, telemetry, operations, analysis, and distribution give the SAMS/PIMS team a unique perspective on space data systems. In 2005, the SAMS/PIMS team was asked to look into generalizing their data system and combining it with the nascent medical instrumentation data systems being proposed for ISS and beyond, specifically the Medical Computer Interface Adapter (MCIA) project. The SpreadRecorder software is a prototype system developed by SAMS/PIMS to explore ways of meeting the needs of both the medical and microgravity measurement communities. It is hoped that the system is general enough to be used for many other purposes.

14. SUBJECT TERMS

Distributed processing; Client server systems; Computer systems design; Computer networks

15. NUMBER OF PAGES

33

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	