A Users Guide to Spread
Version 0.11

Jonathan R. Stanton
jonathan@cnds.jhu.edu

October 21, 2002

Contents

Chapter 1

Introduction to Spread

1.1 What is Spread?

When designing distributed applications one must make a number of architectural choices. These choices include how communication between applications will be handled, what the roles of each process will be, how dependent each machine is on the others for operation of the application, etc. Part of what makes creating reliable, high-performance, useful distributed applications hard is the number of fundamental choices one must make and the complex interactions between each choice.

The group communication model is a framework that provides both a physical toolkit upon which to build and a model which limits the number of choices that must be met. This model simplifies the task of constructing a reliable, correct distributed application while still giving the user a powerful set of abstractions upon which many different distributed applications can be built. It is certainly true that not every application can be built using the group communication model, and even if it could the negative characteristics of the model can make group communications a bad choice. What group communications does do, however, is make a large number of distributed applications easier to build and more powerful. It is no different than any other higher level abstraction.

For example, one could build every network application by creating IP level packets by hand, having the application provide packet checksums, multiplexing, reliability, ordering, and flow control, but everyone realizes that although that is the most powerful approach (and it is used for some specialized applications) in almost every case you want to use a high level API like sockets and an established network protocol like TCP.

The basic services provided by most group communication systems are:

1. Abstraction of a Group (a name representing a set of processes, all of whom receive any messages sent to the Group).

2. Multicast of messages to a Group.

3. Membership of a Group.
4. Reliable messages to a Group.
5. Ordering of messages sent to a Group.
6. Failure detection of members of the Group.
7. A strong semantic model of how messages are handled when changes to the Group membership occur.

It should be obvious that the name "Group Communications System" is very appropriate, as the concept of a "Group" is the fundamental abstraction of the system. Once you have that abstraction all the other services make sense: knowing who is in the group, talking to the group, knowing when someone leaves the group, agreeing on an ordering of events in the group.

Here are a few distinct example applications that exhibit how the group communication model provides a useful abstraction for a wide variety of distributed applications.

- Service and machine monitoring. A number of machines export their status to groups of interested monitors. Whenever failure occurs the monitors are notified.
- Collaborative tools. Many different groups of participants each want to share data, video and audio conferencing.
- DSM (Distributed Shared Memory). Sending pages of memory to machines where it is needed using reliable multicast.
- Highly reliable services (such as air traffic control systems, stock exchanges, military tracking and combat control systems). Services that involve communication of information among numerous machines and people and have high requirements for both availability and fault-tolerance.
- Replicated databases. A number of instances of a database exist in several different locations. They must all be kept synchronized in such a way that a client can query or update any of them and the results will be the same as if only one copy existed.

1.2 Design Issues

1.2.1 Comparison with reliable IP-multicast

The service provided by *Spread* and the service provided by many reliable IP-multicast protocols have some features in common and some differing semantics. The main area of overlap is that they both solve the problem of getting

best-effort reliability when sending multicast messages for small to medium sized groups. The key difference is that most reliable IP-multicast protocols aim to be also solve that problem for very large groups, while *Spread* does not support very large groups, but does provide a stronger model of reliability and additional service such as ordering.

A practical difference is that reliable IP-multicast usually relies on a wide area IP-multicast network (such as the mbone, or ISP support for multicast routing) while *Spread* only relies on point-to-point unicast IP support, and uses IP-multicast only as a performance optimization.

One subtle distinction between reliable IP-multicast and *Spread* 's Reliable service is that *Spread* integrates a membership notification service into the stream of messages. The membership notifications provide some knowledge of who actually received the reliable messages. The issue of membership is a key distinction between the unicast, or point-to-point world of TCP/IP and multicast services. In multicast it is often necessary to know "with whom" you are reliably communicating since there is no obvious 'other party' as in unicast.

1.2.2 Flexibility of services

The key question is at what level of granularity do you define services? GCS allow a number of different levels of service and the application only pays for those that it needs (to a large degree). The GCS primitives are very flexible and many different applications can use them in different ways.

The goal of *Spread* (not necessarily all GCS) is to support the broad middle of applications. This includes those that need more than unreliable multicast or multicast to millions of users, but don't have extremely specialized needs such as hard real-time requirements, hardware fault-tolerance, or esoteric reliability and semantic models. The ideas of GCS have been extended to some of these extremes (especially real-time and hardware assisted fault-tolerance), and have influenced to a small degree the solutions being proposed to reliable multicast to millions of users.

A number of people assert that it is an accepted truth that no one system or protocol will work for all cases. This is essentially a truism. However, they often mean by this that NO system or protocol will be very good for more then one very narrow set of needs, and thus no one should even try to create a system to support many different families of applications. I believe that to be a false assumption because I have seen all of the applications listed above built using one group communication system. All the applications built in this way have fulfilled their requirements and performed well. That is not to say they could not have been built in other ways that might work even better, but they did everything they needed to and because of the standard abstractions and the support of an existing toolkit were able to be built much faster and with more reliability. In essence, the costs of custom designing the services that they needed instead of using the

existing group communication abstractions and services would have been much higher and the performance payoff would not be enough to overcome that.

1.2.3 Modularity of Spread architecture

Spread is designed to be modular in two ways. First, at the network communication level, *Spread* supports multiple link protocols. Second, *Spread* supports multiple client interfaces.

Group communication toolkits can be used in many different environments with very different network infrastructures. These different networks can have very different characteristics (latency, bandwidth, shared/point-to-point, native multicast, routed). *Spread* has a modular API for link protocols that allow different protocols to be used for dissemination, reliability, and flow control without changing the upper layer protocols at all. For example, *Spread* currently has three link protocols implemented in the base system. The first is called the Ring protocol and it provides high throughput when used on a low latency local area network of no more then about 30 daemons. The second uses TCP for transport and provides stable transport over wide area networks in a point-to-point manner. The third is called Hop and like TCP is used to cross wide area networks with high latency and non-negligible loss, but it provides higher throughput and lower message latency than TCP, and is more stable in high loss situations.

The client interfaces provided with Spread include native interfaces for Java and C, and a Perl library that wraps the C interface. These interfaces are designed to be consistent with the language's normal idioms. *Spread* natively only provides a toolkit level abstraction of group communication services. Higher level group tools such as replication tools, wrappers of native networking interfaces, and client specific tools can be implemented on top of the toolkit APIs. One detail is that *Spread* natively supports the Extended Virtual Synchrony(EVS) model (more details on this are later), however another similar model is also very common, the Virtual or View Synchrony model. To support either, *Spread* provides a special client library which implements View Synchrony on top of *Spread* 's native EVS model.

1.3 Spread Guarantees

Spread provides several different types of messaging services to applications. In addition to being able to send messages to entire groups of recipients and receiving membership information about who is currently alive and reachable, *Spread* provides both ordering and reliability guarantees.

When an application sends a *Spread* message it chooses a level of service for that message. The level of service selected controls what kind of ordering and reliability are provided to that message. The application can choose a different

level of service for each message that it sends. *Spread* supports 5 different levels of service. Table 1.1 shows the different types and what kind of ordering and reliability guarantees they provide.

Spread Service Type	Ordering	Reliability
UNRELIABLE_MESS	None	Unreliable
RELIABLE_MESS	None	Reliable
FIFO_MESS	Fifo by Sender	Reliable
CAUSAL_MESS	Causal (Lamport)	Reliable
AGREED_MESS	Total Order (Consistent w/Causal)	Reliable
SAFE_MESS	Total Order	Safe

Table 1.1: Spread Message Service Types

1.3.1 Ordering

The ordering guarantees defined by *Spread* are:

None No ordering guarantee. Any other message also sent with ordering "None" can arrive either before or after this one. Messages with stricter ordering CAN depend on this message. For example, if a FIFO_MESS message M_a follows RELIABLE_MESS message M_b then M_a cannot be delivered until M_a has been delivered (but the reverse is not true).

Fifo by Sender All messages sent by this sender [1] of at least Fifo ordering are delivered in FIFO order. As mentioned above a RELIABLE_MESS sent after a Fifo message may be delivered before the Fifo message.

Causal (Lamport) All messages sent by all senders are delivered in an order consistent with Lamport's definition of "Causal" order. This order is consistent with Fifo ordering.

Total Order (Consistent w/Causal All messages sent by all senders are delivered in the exact same order to all recipients. This order is also consistent with Causal order. It is provided by making the partial order defined by causal into a total order. The total order uses the id of the sender to break ties.

It is important to note that messages sent with Fifo ordering or less do not support the full membership semantics of *Spread* . This is a result of *Spread* optimizing two common operations, group joins and leaves and sending FIFO or Reliable

[1] A sender is defined as a particular connection to a *Spread* daemon, so an application with 3 connections will be considered 3 different senders

messages. First, joins and leaves of group members do not cost more then sending one SAFE message and result in no extra synchronization costs. Second, Fifo and Reliable messages are not delayed before delivery by any other messages. So even if gaps exist in the global order of all messages, Reliable messages can still be delivered and Fifo messages can be delivered as long as all the messages from their sender have arrived. Because of these two optimizations, it is possible for a Reliable or Fifo message to be delivered earlier then it would be if it was globally ordered, however a gap in the global sequence may contain a join or leave message (since they are just SAFE messages) so it might be that one process delivers the Fifo or Reliable message before the join and a different process delivers the join first and then the message.

1.3.2 Reliability

The Reliability guarantees defined by *Spread* are:

Unreliable The message is unreliable. It may be dropped or lost and will not be recovered by *Spread* .

Reliable The message will be reliably delivered to all recipients who are members of the group to which the message was sent. *Spread* will recover the message to overcome any network losses.

Safe The message will ONLY be delivered to a recipient if the daemon that recipient is connected to knows that all *Spread* daemons have the message. If a membership change occurs, and as a result the daemon cannot determine whether all daemons in the old membership have the message, then the daemon will deliver the Safe message after a TRANSITIONAL MEMBERSHIP message.

1.4 Additional Information

Spread is actively developed by the Center for Networking and Distributed Systems at Johns Hopkins University. The software, documentation, community of users, and additional applications are constantly being improved and evolving. The best way to find out what is currently going on, or learn more about the *Spread* system is to check out our web sites:
`http://www.cnds.jhu.edu/`
`http://www.spread.org/`

A number of research papers have been published on the *Spread* system and related projects. A complete list can be found on the web.

Chapter 2

Installing and Configuring Spread

2.1 Installing Spread

Spread is a fairly simple software package to install. The runtime components are just one executable called `spread` and a configuration file located somewhere the *Spread* daemon can find it. Developing *Spread* applications requires also installing two libraries and some header files. Finally, the *Spread* documentation consists of a set of man pages which are installed in the usual location and some other documentation files, such as README files, html documentation, and online books, which can be installed wherever is convenient.

2.1.1 Downloading

Spread can be downloaded from `http://www.spread.org/` or `http://www.cnds.jhu.edu/`.

2.1.2 Installing a binary distribution

We recommend that if you are experimenting with spread you create a special 'spread' directory (for example /usr/local/spread or /opt/spread) and keep all the files together there so things are easier to find. This also makes it easier to run multiple architectures as the binaries for each are in their own subdirectory. This is not necessary though. You can create that directory anywhere (e.g. your own directory).

If you are installing spread for active use it is probably easier to just install the correct version of the binaries, headers, man pages, and libraries into your standard locations. The directions below assume you are doing this.

1. Unpack the spread.tar.gz file into a temporary directory
2. Look at the Readme for any updates

3. Select the appropriate architecture:

```
arch-bsdi
arch-sgi
arch-sunos
arch-sunsol
arch-pcsol
arch-linux
arch-freebsd
```

4. Type 'make arch-????' with your architecture as the option to make.

5. Now you need to copy the files, I will assume you use /usr/local/bin,include,lib,man. Replace "ARCH" with the directory for your architecture.

```
cp -p include/* /usr/local/include/
cp -p ARCH/libspread.a /usr/local/lib/
cp -p ARCH/libtspread.a /usr/local/lib/
cp -p ARCH/spread /usr/local/bin/
cp -p ARCH/monitor /usr/local/bin/
cp -p ARCH/user /usr/local/bin/
cp -p ARCH/tuser /usr/local/bin/
cp -p ARCH/simple_user /usr/local/bin/
cp -p ARCH/flooder /usr/local/bin/
cp -p docs/*.3 /usr/local/man/man3/
cp -p docs/*.1 /usr/local/man/man1/
```

6. To run *Spread* you need a configuration file. See Section 2.2.

To use the Java classes and examples you need to have a copy of the main 'spread' daemon running. Then the spread/*.class files gives you the equivalent of the libspread.a as a package of java classes. The user.java, user.html, and user.class files give you a demonstration applet and source code. The tree.html AllNames.html and packages.html give some documentation for the java interface.

For Windows (95/NT) systems use the spread.exe daemon and the libspread.lib or libtspread.lib to link with your programs.

2.1.3 Installing a source distribution

The source install is uses the standard autoconf and make tools on Unix like systems and a set of Visual C++ project files on Windows. Generally Spread should build on almost any Unix-like system or any other OS that has standard BSD socket support. See the file PORTING for hints on porting.

From the directory where you unpacked the Spread source distribution do the following:

1. Run "./configure" If you want the binaries and libraries to be installed somewhere other then /usr/local/, pass configure a –prefix=/my/location/path option.

2. Run "make"

3. If you want to install the binaries into your standard system locations, change to a privledged user and run "make install" Otherwise, run "make install" as the user you want to install Spread as.

2.2 Configuring Spread

Spread requires some configuration to be able to run. The daemons rely on a configuration file to both set any runtime variables and to specify the location of all the other potential daemons in the network.

2.2.1 Planning a Spread Network

A sample configuration is found in Fig 2.1 for a one site *Spread* network and in Fig 2.2 for a two site *Spread* network where the sites are directly connected (i.e. they do not have to cross the public Internet). Fig 2.3 shows the most generic network configuration where there are several local area networks connected over the Internet.

In Fig 2.1 the sample IP addresses are from the reserved private address space (192.168.*.*). This is ONLY possible in *Spread* when all the applications that will connect to *Spread* are also on machines located in the same private IP address space.[1] Normally, all interfaces used by *Spread* need to have public, valid IP addresses. For the rest of the examples I use real addresses which are missing the first byte (it is replaced by an x).

2.2.2 Creating a Configuration File

A good starting point for your configuration file is the sample file, called `sample.spread.conf` included with all distributions of spread. The file is commented and gives basic instructions on modifying it. The configuration file can be located in the current working directory from which you run the `spread` executable, it can be located

[1] It might be possible to have a masquerading router in front of the entire cluster where the router re-maps external client TCP connections to the internal private IP addresses of the servers. This architecture is probably only useful in a certain limited class of applications.

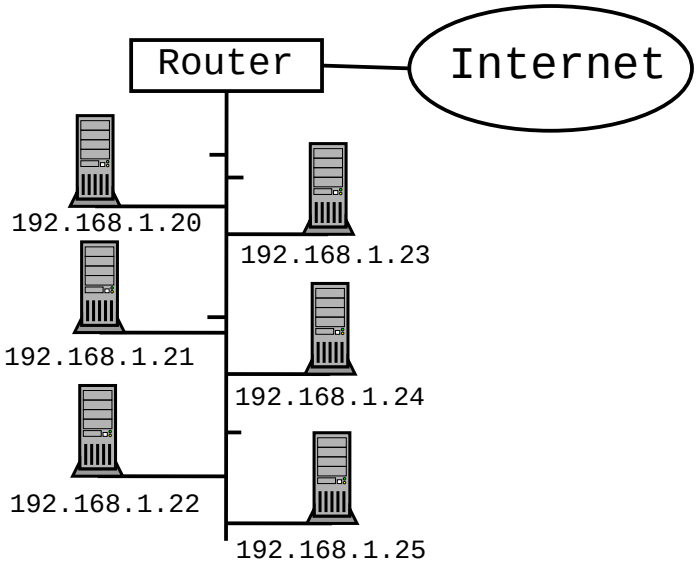

Figure 2.1: Sample Network with one site

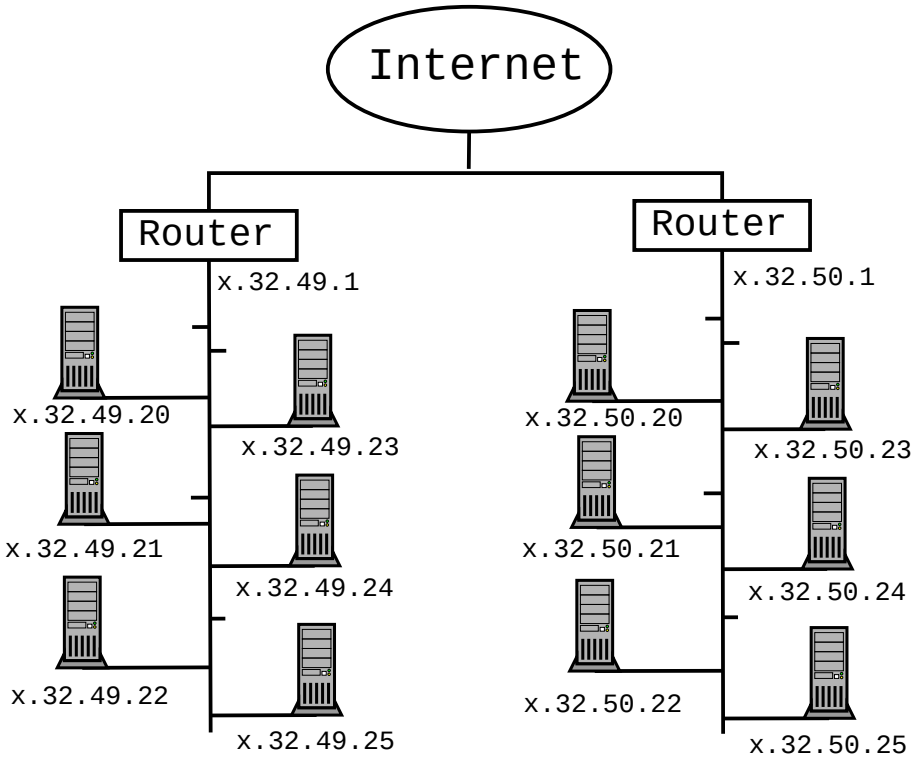

Figure 2.2: Sample Network with two sites directly connected

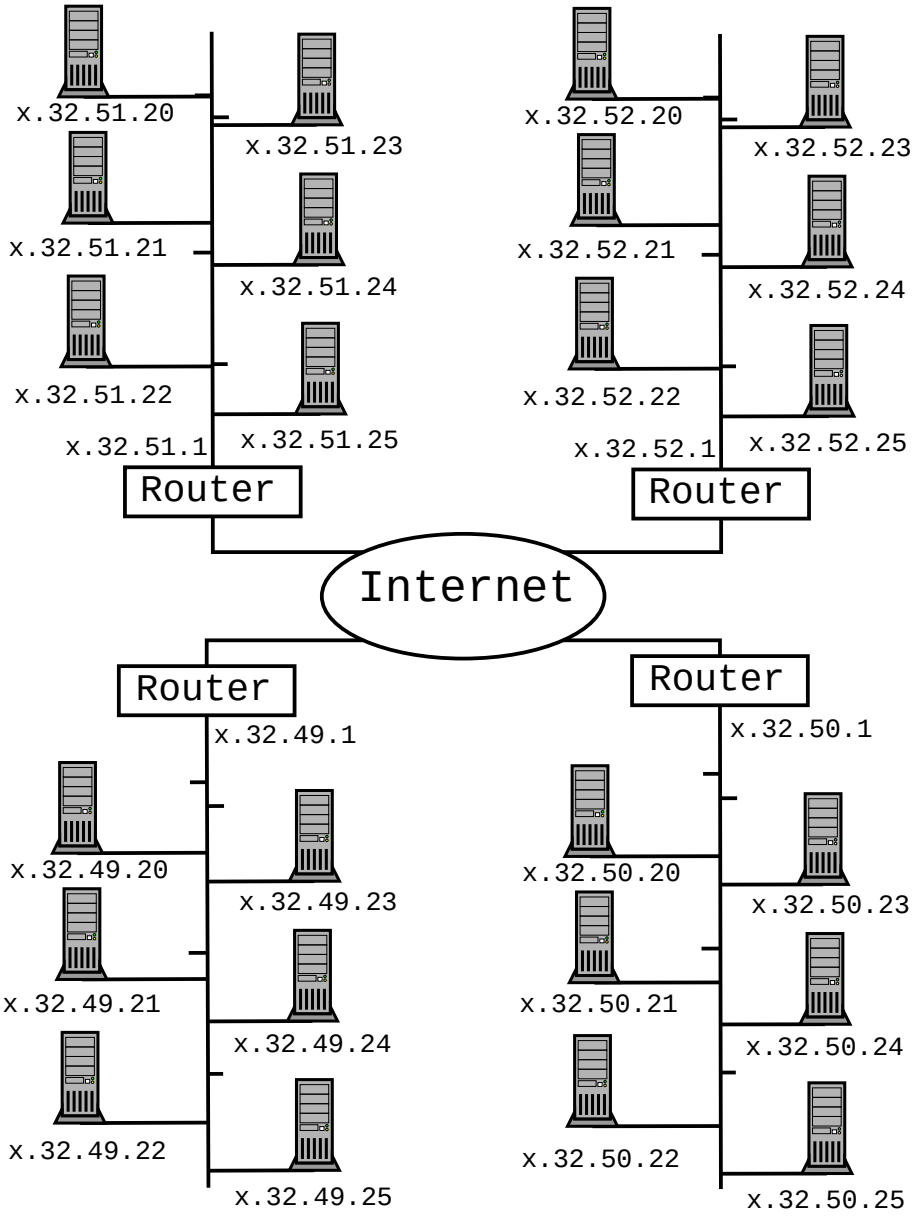

Figure 2.3: Sample Network with four sites connected over the Internet

```
Spread_Segment 192.168.1.255:3333 {
    machine1    192.168.1.20
    machine2    192.168.1.21
    machine3    192.168.1.22
    machine4    192.168.1.23
    machine5    192.168.1.24
    machine6    192.168.1.25
}
```

Figure 2.4: Sample configuration file for one site

```
Spread_Segment  x.32.49.255:3333 {
    machine1    x.32.49.20
    machine2    x.32.49.21
    machine3    x.32.49.22
    machine4    x.32.49.23
    machine5    x.32.49.24
    machine6    x.32.49.25
}
Spread_Segment  x.32.50.255:3333 {
    machineB1   x.32.50.20
    machineB2   x.32.50.21
    machineB3   x.32.50.22
    machineB4   x.32.50.23
    machineB5   x.32.50.24
    machineB6   x.32.50.25
}
```

Figure 2.5: Sample configuration file for two sites directly connected

in /etc/, or it can be located anywhere and passed to the `spread` executable on the command line with the `-c` option.

Some sample configuration files which are based on the sample network configurations described earlier are provided here in Figure 2.4, Figure 2.5 and Figure 2.6.

The configuration file can also contain option commands which allow the user to change the daemon's behavior at runtime. These options are shown in Figure 2.7. The options can appear anywhere in the configuration file outside of a Spread_Segment.

The `DebugFlags` option controls what logging and activity information *Spread* provides as it runs. Table 2.1 shows the available flags and what they do. The flags can also be negated by the '!' character, so the flags

```
DebugFlags = { ALL !DATA_LINK !EVENTS }
```

will print all log messages except those related to data-link or events. The PRINT and EXIT flags should always be enabled for correct operation of *Spread* .

The log messages are either printed to the screen of the console where *Spread* is run or to the log file specified by the `EventLogFile` option. The `EventLogFile`

```
Spread_Segment  x.32.49.255:3333 {
    machine1    x.32.49.20
    machine2    x.32.49.21
    machine3    x.32.49.22
    machine4    x.32.49.23
    machine5    x.32.49.24
    machine6    x.32.49.25
}
Spread_Segment  x.32.50.255:3333 {
    machineB1   x.32.50.20
    machineB2   x.32.50.21
    machineB3   x.32.50.22
    machineB4   x.32.50.23
    machineB5   x.32.50.24
    machineB6   x.32.50.25
}
Spread_Segment  x.32.51.255:3333 {
    machineC1   x.32.51.20
    machineC2   x.32.51.21
    machineC3   x.32.51.22
    machineC4   x.32.51.23
    machineC5   x.32.51.24
    machineC6   x.32.51.25
}
Spread_Segment  x.32.52.255:3333 {
    machineD1   x.32.52.20
    machineD2   x.32.52.21
    machineD3   x.32.52.22
    machineD4   x.32.52.23
    machineD5   x.32.52.24
    machineD6   x.32.52.25
}
```

Figure 2.6: Sample configuration file for four sites connected by the Internet

```
DebugFlags = { PRINT EXIT }
EventLogFile = testlog.out
#EventLogFile = spread_%h.log
EventTimeStamp = "[%a %d %b %Y %H:%M:%S]"
DangerousMonitor = false
#SocketPortReuse = AUTO
#RuntimeDir = /var/run/spread
#DaemonUser = spread
#DaemonGroup = spread
```

Figure 2.7: Sample configuration file options

Flag	Function
PRINT	General info that should always be printed.
EXIT	Errors or other events that cause Spread to quit.
DEBUG	Debugging information.
DATA_LINK	Lowest level of sending and receiving datagrams.
NETWORK	Packing messages and setting who to talk with.
PROTOCOL	Ordering, Token handling, and delivery algorithms.
SESSION	Per user connection management.
CONFIGURATION	Parsing and loading configuration file.
MEMBERSHIP	State and messages sent during membership changes.
FLOW_CONTROL	Flow control state of the ring.
STATUS	Reporting of status information to the monitor.
EVENTS	All events (timed, fd based) and main loop.
GROUPS	Group state and group membership changes.
MEMORY	Memory debugging and allocation.
SKIPLIST	State of data structure.
ALL	Enables all flags.
NONE	Disables all flags.

Table 2.1: Available Debug flags for configuration file

filename can contain the special string `'%h'` which will be replaced with the hostname of the machine running the *Spread* daemon. This makes it easy to have one configuration file which multiple daemons will use from the same NFS mounted filesystem. An example is shown in Figure 2.7 in line 3.

The log messages will be prefixed with a timestamp string if the `EventTimeStamp` option is enabled. The timestamp has a default format similar to most log timestamps. The format can be customized by setting `EventTimeStamp` equal to a format string as shown in Figure 2.7.

The SocketPortReuse option allows one to choose when the SOREUSEADDR socket option is used on TCP sockets opened up by Spread. When a TCP socket is open in a server and clients are connected, if the server crashes or goes down without cleanly closing off all of the client TCP connections, some connections can be left in "TIME_WAIT" state on the server which will prevent the server from restarting (the bind to the TCP socket will fail) for about 2 minutes (the timeout on TIME_WAIT state). In an environment where you desire to restart the servers immediately in the event of a crash or shutdown, this 2 minute wait is clearly undesirable. The SOREUSEADDR socket option allows the daemon to restart immediately, even if some connections are still in TIME_WAIT state. However, as a consequence of how it does this, it may also allow OTHER programs to bind to the same port number and interface as Spread is bound to and possibly steal the messages destined for Spread.

This is a potentially serious security issue as it could allow a user who has access to the machine running the Spread daemon to capture Spread traffic, or interfere with the correct functioning of Spread. This security issue is well known in the Operating system and Internet community and a number of operating systems have modified the SOREUSEADDR option to minimize of avoid the security issues while maintaining it's useful properties. So in many cases it is safe to enable the SocketPortReuse option, and the default that Spread ships with is AUTO. In AUTO setting the SORESUSEADDR option is enabled when Spread is configured in the spread.conf file to only bind to specific interfaces, and is disabled when Spread binds to INADDR_ANY (where no specific interfaces are specified in the spread.conf file). We believe this is a safe option as the security issue only arises when a program binds to INADDR_ANY.

If you know you are running on an operating system which has a secure implementation of SOREUSEADDR, or you do not allow any non-trusted users to run programs on the same machines as Spread daemons run on, you can set this to "On" and the daemon will always use this option to allow fast restarts. If you want to disable this option completely so the daemon will *Never* use the SOREUSEADDR option, set this to "Off".

The RuntimeDir, DaemonUser and DaemonGroup options allow runtime configuration of the file system location and uid/gid combination that Spread runs itself as when it is executed with root privledges. This option only applies to Unix-like operating systems. When executed with root privledges, Spread will change to the specified directory in the file system and use the `chroot` system call to change it's `/` directory to be that directory. It will then drop all of it's privledges and continue to run as the user and group specified. Spread does both of these actions *after* reading in it's config file and opening the specified log file, so both files can exist outside of the RuntimeDir directory. No files need to be installed in this directory tree.

2.3 Running the Daemon and Clients

To get help on running any program distributed with *Spread* , just type the programs name followed by `usage`. For example to if you run the command:

```
> ./spuser usage
Usage: spuser
        [-u <user name>]  : unique (in this machine) user name
        [-s <address>]    : either port or port@machine
```

you get the command line options to the `spuser` program.

The `spread` executable is usually run in the background with standard output and error redirected to `/dev/null` on Unix machines.

2.3.1 Running the Monitor

The `spmonitor` program can be useful for gathering information about how the *Spread* daemons are working and detecting some problems. It can also be used as a management tool to terminate daemons or adjust their flow control parameters.

As a basic security feature the daemons will only accept commands from a monitor process which is run on a machine in the configuration file used by the daemon. Since monitor commands include the ability to tell the daemon to quit, at least this type of restriction is required. Currently, the way the restriction is implemented only stops someone who does not try to impersonate a different source IP address.

The recommended way to protect your *Spread* cluster from rouge monitors is to set the `DangerousMonitor` option to `false` in the `spread.conf` file. Then the partition, kill and flow_control commands in the monitor are disabled on the daemon side. So the daemon will ignore any monitor requests to do those actions no matter where they come from.

The `DangerousMonitor` setting is defaulted to FALSE in version 3.13. If you need to use the partition, flow_control or kill commands you will need to enable them by setting `DangerousMonitor` to `true` in your configuration file.

The monitor can take three command line options as shown below.

```
Usage: spmonitor
    [-p <port number>]: specify port number
    [-n <proc name>]  : force computer name
    [-t <status timeout>]: specify number of seconds between status queries
    [-c <file name>]  : specify configuration file
```

The `spmonitor` program will look for a file called `spread.conf` in three locations: first, wherever the `-c` command line option give if it is used, second in the directory it is started from, and third in /etc/.

Once it has loaded the configuration file, monitor will give a brief text menu and prompt as shown in figure 2.8. You can then select what you want to do and to which daemons you want the command sent. The most common command will be to send a daemon a status query. The results of that query will look something like Figure 2.9. Some of the more interesting and useful information returned in this status report are:

1. Line 1: The state and gstate should both be 1 during normal operation. Other values indicate a membership change is occuring.

2. Line 1: The “after 116 seconds” gives the time this daemon has been alive.

3. Line 2: Gives the total number of alive daemons and how many different segments they are in.

4. Line 3: The rounds value is the number of times the token has revolved around the daemons.

```
=============
Monitor Menu:
-------------
        0. Activate/Deactivate Status {all, none, Proc, CR}

        1. Define Partition
        2. Send   Partition
        3. Review Partition
        4. Cancel Partition Effects

        5. Define Flow Control
        6. Send   Flow Control
        7. Review Flow Control

        8. Terminate Spread Daemons {all, none, Proc, CR}

        9. Exit

Monitor>
```

Figure 2.8: Monitor menus

```
Status at tesseract V3.13 (state 1, gstate 1) after 116 seconds :
Membership  :  1  procs in 1 segments, leader is tesseract
rounds   :    1598      tok_hurry :      66     memb change:       1
sent pack:      14      recv pack :       0     retrans    :       0
u retrans:       0      s retrans :       0     b retrans  :       0
My_aru   :      14      Aru       :      14     Highest seq:      14
Sessions :       0      Groups    :       0     Window     :      60
Deliver M:      12      Deliver Pk:      14     Pers Window:      15
Delta Mes:       0      Delta Pack:       0     Delta sec  :      10
==================================
```

Figure 2.9: Monitor Status Report

5. Line 4: Sent pack and recv pack give the cumulative number of actual packets sent or received.

6. Line 4/5: A large number of retransmissions of any time (retrans = u retrans + s retrans + b retrans) is obviously bad.

7. Line 7: The Sessions is the number of locally connected clients.

8. Line 7: The Groups is the total number of groups that currently exist in the system.

9. Line 7/8: Window is the flow control window limiting how many packets are sent each token revolution, Pers Window limits each daemon from initiating more then that number of packets each time it gets the token.

10. Line 8: Deliver M is really "Deliver Messages" and is the cumulative total number of messages this daemon has been able to deliver.

11. Line 8: Deliver Pk is really "Deliver Packets" and is the cumulative total number of packets (a message may contain multiple packets) this daemon has delivered.

12. Line 9: Delta Mess and Delta Pack are the changes in the delivered message and packet counters since the last status query was sent.

13. Line 10: Delta sec gives the time between this status query and the last one.

2.4 Tuning Spread for Performance or Unique Situations

The binary distribution and the source code as distributed are tuned to work under almost any situation and networking environment. As a result, they are not tuned for the highest performance, the fastest fail-over, or the most scalability. To use *Spread* most effectively in production environments some tuning is often necessary.

This section will address general tuning information first and then provide solutions to several standard problems that we have encountered.

2.4.1 Membership timeouts

The default *Spread* membership algorithm uses several timeout values to determine how long to wait before determining a failure has occured, how long to keep searching for more members during a change, and how often to look for new members. There are two default sets of values, one is used when the configuration

only includes one segment and the other is used when more then one segment are currently active. The current values are shown below from lines 128-151 of membership.c.

```
        if( Wide_network )
        {
                Token_timeout.sec  =  20; Token_timeout.usec  = 0;
                Hurry_timeout.sec  =   6; Hurry_timeout.usec  = 0;

                Alive_timeout.sec  =   1; Alive_timeout.usec  = 0;
                Join_timeout.sec   =   1; Join_timeout.usec   = 0;
                Rep_timeout.sec    =   5; Rep_timeout.usec    = 0;
                Seg_timeout.sec    =   2; Seg_timeout.usec    = 0;
                Gather_timeout.sec =  10; Gather_timeout.usec = 0;
                Form_timeout.sec   =  10; Form_timeout.usec   = 0;
                Lookup_timeout.sec =  90; Lookup_timeout.usec = 0;
        }else{
                Token_timeout.sec  =   5; Token_timeout.usec  = 0;
                Hurry_timeout.sec  =   2; Hurry_timeout.usec  = 0;

                Alive_timeout.sec  =   1; Alive_timeout.usec  = 0;
                Join_timeout.sec   =   1; Join_timeout.usec   = 0;
                Rep_timeout.sec    =   2; Rep_timeout.usec    = 500000;
                Seg_timeout.sec    =   2; Seg_timeout.usec    = 0;
                Gather_timeout.sec =   5; Gather_timeout.usec = 0;
                Form_timeout.sec   =   5; Form_timeout.usec   = 0;
                Lookup_timeout.sec =  60; Lookup_timeout.usec = 0;
        }
```

In a small local area configuration of daemons these values can be decreased significantly to improve fault detection time as long as the proportions are kept the same and a few other constrainst are maintained. Basically, the smallest timeout should not be less than twice the standard kernel scheduling delay (often 10-20ms) plus twice the packet latency of the network. Also, the smallest timeout should increase as the number of daemons increases.

The Token, Gather, and Form timeouts should be sufficient to allow a full rotation of the token (so each machine gets it and has time to do some work while holding it) plus some slack for an occasional retransmission or delay. You do not want to trigger token loss timeouts without being sure you really lost it because they have a significant cost in reforming an membership. The Lookup timeout determines how often the leader of an active membership probes to find other possible active daemons who are not part of the leader's current membership.

2.4.2 Spread in high-load environments

If the machines running the daemons are heavily loaded (say with loads of 10 or higher) it is more difficult for *Spread* to work well. This is because *Spread* is performing real-time routing and messaging as a user-level program. When the load is high *Spread* is able to be scheduled onto the CPU less and less often. Since all the daemons in the membership rely on all the others to quickly process the token, send out new packets and then forward the token on, if even one machine is

heavily loaded the token will be significantly slowed and all *Spread* daemons will be slowed. When the load gets very high (over 30, or less on a large configuration), these delays can even cause spurious membership changes as the daemons think the token was lost, even though it is just slow because of the delays.

The best solution so far for this situation is to make some of the following three changes.

First, modify the timeouts as described in Section 2.4.1 to be larger. Especially increase the Token_timeout and Form_timeout to be at least several seconds larger then the longest average time a token takes to get to all the machines. For example, if because of scheduling delays each daemon takes 300 ms to get the cpu when a token arrives, then allow at least 350 ms per daemon. So with 30 machines you will want 11 seconds plus a few so maybe 15 second timeout for the Token. One way to calculate this delay is to run the monitor and query one machine every second watching the token-rounds variable. See how many seconds it takes for one round of the token to occur under the highest load you normally experience. Then add a few seconds and use that as your timeout.

Second, run the *Spread* daemon with real-time scheduling priority.[2] This is standard on all unixes (and can also be done on Windows), and is quite simple. This will give *Spread* the first chance at the CPU whenever it needs it. The costs of this are straightforward. First it requires root privilege on the machine the daemon runs on, and second if *Spread* for some reason becomes a runaway process not releasing the CPU it is impossible to stop unless you also have a shell set to a higher real-time scheduling priority. We have never seen *Spread* runaway with CPU and it is very unlikely a bug could cause it because of the event based design of *Spread* .

Third, use the monitor to adjust the flow control parameters of the token. Since each token rotates much slower under high load then under light load the daemons are sending fewer messages per second on the network. If the load is high but spare bandwidth on the network is available, you could try increasing the number of packets each daemon is allowed to send when it gets the token (the Personal_window of each daemon) and the total number of packets that can be sent during each rotation of the token (the Window).

2.4.3 Dealing with bursty application traffic

Sometimes an application will have a bursty pattern of generating traffic. This has many possible causes, but the result is that you want the *Spread* daemon to accept somewhat more messages then it can really handle at any one time. Of course, if you continually send faster then the underlying network and *Spread* can handle the daemon will have to block the senders, but when the average rate is manageable

[2] A small program to give the daemon real-time scheduling priority can be found at our web site http://www.spread.org/software/

but bursts are higher some buffering by the daemon can help significantly.

Two specific values can be of use in tuning the available buffering. The first is the WATER_MARK variable defined in the `spread_params.h` file. This sets the number of messages *Spread* will accept from all client connections, without sending them on, before blocking the applications. Once *Spread* has actually sent some of the messages onto the network it will unblock the applications.

The second is the number of buffers that *Spread* will keep for each receiver when delivering messages. If the client application is not calling SP_receive sufficiently often to keep up with the number of messages being delivered to it then *Spread* will buffer upto MAX_SESSION_MESSAGES.

2.4.4 Changing the number of daemons per segment

By default *Spread* is compiled to support a maximum of 128 machines per segment and up to 20 segments with a total of 128 machines at most in a configuration. The 128 machine limit is a hard limit and the protocol has never been tested with more than 50-60 daemons. The parameters that control these limits are `MAX_PROCS_SEGMENT, MAX_SEGMENTS, MAX_PROCS_RING` in `spread_params.h`. `MAX_PROCS_RING` can never be more than 128, so there is not much point in changing it. However if you want to only allow one big segment you could change `MAX_SEGMENTS` to 1. Or if you want more then 20 segments (say 25), each of which has only a few machines, you could set `MAX_SEGMENTS` to 25 and `MAX_PROCS_SEGMENT` to 10.[3]

Spread will perform best when the configuration of machines into segments matches the actual network configuration of the machines because *Spread* assumes that machines in one segment can be reached by a single broadcast and have very low latency while multi-segment configurations are assumed to have higher latency between segments and to not all be reachable by a broadcast. If a collection of machines who are actually on one Ethernet segment are divided into several *Spread* segments then each data message sent will be sent multiple times on the physical network, which is more inefficient than necessary.

[3]The only advantage of shrinking the `MAX_PROCS_SEGMENT` is a small decrease in the required memory so in almost all cases you will not need to change this value.

Chapter 3

Spread C API

3.1 Introduction

3.1.1 Short Buffer Handling

It is the traditional behavior of networking APIs that when a user provided buffer is insufficient, the API will provide as much data as possible and truncate the rest. Sometimes the user receives a notice that some data was truncated and sometimes no notification is given. Thus it is the user's responsiblity to detect when datagrams are too short and recover in some way (such as re-requesting data).

The difficulty with using this approach in *Spread* is that when the application has to recover from this some properties of the message are lost. For example, if the message was a SAFE message, the other members can rightly assume that either all the members will get the data or they will not get it because they crash or disconnect from *Spread* . In this case some members might get part of the data, but have to recover the rest of it, also the data can be lost even when the process continues to execute correctly which makes it difficult for the other members to detect the fault.

Essentially because each message has attached meaning, such as ordering, or reliability guarantees, unpredictable loss of data in an otherwise reliable system compromises the very semantics we want to use. It is possible to check for this loss and recover, but the costs are significant, especially when weighed against the cost of avoiding the problem in the first place. Thus, unlike UDP datagrams, *Spread* messages are designed to be reliable even with short buffers.

The method used is straightforward. *Spread* will never truncate large messages unless you explicitly ask it to. When you call SP_receive with a data buffer or groups list too short to hold all the data, the SP_receive function will return with an error code of GROUPS_TOO_SHORT or BUFFER_TOO_SHORT and *NO* data or groups will be returned. The only information that will be returned is in the following parameters:

`service_type`	set to the correct type for the message.
`sender`	is empty.
`num_groups`	set to the number of groups the *groups* parameter needs to accept to avoid a GROUPS_TOO_SHORT error. This number is returned as a negative number. If the *groups* parameter was large enough the *num_groups* field will be 0.
`groups`	is empty.
`mess_type`	set to the message type field the application sent with the original message, this is only a short int (16bits). This value is already endian corrected before the application receives it.
`endian_mismatch`	set to the size, in bytes, of the data buffers needed to completely receive this message and avoid a BUFFERS_TOO_SHORT error. This number is returned as a negative number. If the data buffers provided were large enough then the *endian_mismatch* field will be set to 0.
`mess`	is empty.

So, when SP_receive returns one of the *_TOO_SHORT errors you can examine the *service_type* and *mess_type* fields to get some information about what kind of message *Spread* is trying to give you. You can then examine the *num_groups* and *endian_mismatch* fields to discover how large your buffers need to be. If either field is set to 0 then that buffer was large enough and does not have to be increased. Obviously this can only be true for one of the buffers since one of them was an actual error. You then increase your application buffers and call SP_receive again. It should return with the message and without error (unless something else is also wrong).

This retry approach is safe with multi-threaded applications because each call succeeds or fails on it's own and if two threads retry for the same message, one will get it and the other will get the message after it (which is what would happen anyway if they were not retrying).

The retry approach does, however, *require* that the application check for errors when calling SP_receive and if a *_TOO_SHORT error occurs they either enlarge their buffers or call SP_receive again with the DROP_RECV flag set, as described below. If they either ignore errors or do not correct the short buffers, the application will continually loop calling SP_receive and never receive anything.

If the application does not want to actually receive the entire data buffer or groups list, it has the option of calling SP_receive with the *service_type* field set to the DROP_RECV flag. When this is done, *Spread* will treat the message just like most networking systems and return all the data and groups that will fit in the available space and truncate the rest. It will still return an error value informing the application that it has lost data. In simple applications or ones with relaxed, or

specialized requirements this might be more useful then having to check for error values and retry the SP_receive.

3.2 API Datatypes

The *Spread* API uses only a few specific data types.

```
#define         mailbox                 int
#define         service                 int

#define         MAX_SCATTER_ELEMENTS    100

typedef struct  dummy_scat_element{
        char    *buf;
        int     len;
} scat_element;

typedef struct  dummy_scatter{
        int             num_elements;
        scat_element    elements[MAX_SCATTER_ELEMENTS];
} scatter;

typedef struct  dummy_group_id {
        int32           id[3];
} group_id;
```

3.3 SP_Functions

3.3.1 SP_connect

#include <sp.h>
int SP_connect(const char * *spread_name*, const char * *private_name*, int *priority*, int *group_membership*, mailbox * *mbox*, char * *private_group*);

`SP_connect` is the initial call an application must make to establish a connection with a Spread daemon. All other spread calls must refer to a valid *mbox* set by this function (mbox is passed by reference).

The *spread_name* is the name of the Spread daemon to connect to. It should be a string in one of the following forms:

`4803` connect to the Spread daemon on the local machine using Unix Domain Sockets with socket on /tmp/4803. This form cannot be used to connect to a Windows95/NT machine.

`4803@localhost` connect to the Spread daemon on port 4803 of the local machine through loopback TCP/IP. This form can be used on Windows95/NT machines.

`4803@host.domain.edu`	connect to the machine identified by the domain name "host.domain.edu" on port 4803.
`4803@x.y.221.99`	connect to the machine identified by the IP address "x.y.221.99" on port 4803.

The *private_name* is the name this connection would like to be known as. It must be unique on the machine running the spread daemon. The name can be of at most MAX_PRIVATE_NAME characters with the same character restrictions as a group name (mainly it cannot contain the '#' character).

The *priority* is a 0/1 flag for whether this connection will be a "Priority" connection or not. Currently this has no effect.

The *group_membership* is a boolean integer. If 1 then the application will receive group membership messages for this connection, if 0 then the application will *not* receive *any* membership change messages.

The *mbox* should be a pointer to a mailbox variable. After the `SP_connect` call returns this variable will hold the mbox for the connection.

The *private_group* should be a pointer to a string big enough to hold at least MAX_GROUP_NAME characters. After the `SP_connect` call returns it will contain the private group name of this connection. This group name can be used to send unicast messages to this connection and no one can join this special group.

RETURN VALUES

`ACCEPT_SESSION`	on success.
`ILLEGAL_SPREAD`	*spread_name* given to connect to was illegal for some reason. Usually because it was a unix socket on Windows95/NT, an improper format for a host or an illegal port number
`COULD_NOT_CONNECT`	lower level socket calls failed to allow a connection to the specified spread daemon right now.
`CONNECTION_CLOSED`	during communication to establish the connection errors occured and the setup could not be completed.
`REJECT_VERSION`	the daemon or library has a version mismatch.
`REJECT_NO_NAME`	no user private name was provided.
`REJECT_ILLEGAL_NAME`	name provided violated some requirement (length or used an illegal character)
`REJECT_NOT_UNIQUE`	name provided is not unique on this daemon. Recommended response is to try again with a different name.

3.3.2 SP_disconnect

```
#include <sp.h>
int SP_disconnect( mailbox mbox );
```

`SP_disconnect` should be called when the application is finished with a connection to the Spread daemon. The application may have other connections still open to the daemon and may open a new connection after disconnecting.

The *mbox* should be for the connection you wish to disconnect from.

RETURN VALUES

`NORMAL`	returns 0 on success
`ILLEGAL_SESSION`	when the session *mbox* given is not a valid connection.

3.3.3 SP_join

```
#include <sp.h>
int SP_join( mailbox mbox , const char * group );
```

`SP_join` joins a group with the name passed as the string *group*. If the group does not exist among the Spread daemons it is created, otherwise the existing group with that name is joined.

The *mbox* of the connection upon which to join a group is the first parameter. The *group* string represents the name of the group to join.

RETURN VALUES

`NORMAL`	returns 0 on success.
`ILLEGAL_GROUP`	the *group* given to join was illegal for some reason. Usually because it was of length 0 or length > MAX_GROUP_NAME
`ILLEGAL_SESSION`	the session specified by *mbox* is illegal. Usually because it is not active.
`CONNECTION_CLOSED`	during communication errors occured and the join could not be initiated.

3.3.4 SP_leave

```
#include <sp.h>
int SP_leave( mailbox mbox, const char * group );
```

`SP_leave` leaves a group with the name passed as the string *group*. If the group

does not exist among the Spread daemons this operation is ignored, otherwise the group is left.

The *mbox* of the connection upon which to leave a group is the first parameter. The *group* string represents the name of the group to leave.

RETURN VALUES

`NORMAL`	returns 0 on success.
`ILLEGAL_GROUP`	the *group* given to leave was illegal for some reason. Usually because it was of length 0 or length > MAX_GROUP_NAME
`ILLEGAL_SESSION`	the session specified by *mbox* is illegal. Usually because it is not active.
`CONNECTION_CLOSED`	during communication errors occured and the leave could not be initiated.

3.3.5 SP_multicast and family

```
#include <sp.h>
int   SP_multicast(mailbox mbox,   service service_type,   const char * group,
                   int16 mess_type, int mess_len, const char * mess );
int       SP_scat_multicast(       mailbox mbox,          service service_type,
                   const char * group, int16 mess_type, const scatter scat_mess );
int     SP_multigroup_multicast(mailbox mbox     ,      service service_type,
                   int num_groups   ,   const char groups[][MAX_GROUP_NAME],
                   int16 mess_type, int mess_len, const char * mess );
int     SP_multigroup_scat_multicast(mailbox mbox,      service service_type,
                   int num_groups,     const char groups[][MAX_GROUP_NAME],
                   int16 mess_type, const scatter scat_mess );
```

SP_multicast and its variants all can send a message to one or more groups. The message is sent on a particular connection and is marked as having come from that connection. The *service_type* is a type field that should be set to the service this message requires. The valid flags for messages are:

- UNRELIABLE_MESS
- RELIABLE_MESS
- FIFO_MESS
- CAUSAL_MESS
- AGREED_MESS

- SAFE_MESS

This type can be bit ORed with other flags like SELF_DISCARD if desired. Currently SELF_DISCARD is the only additional flag.

If the `SP_multicast` or `SP_scat_multicast` versions are being used then only one group can be sent to. So the *group* string should include the name of the group to send to. If a multigroup varient is being used, then the groups are specified by the *num_groups* integer and the array of group names called *groups* representing all the groups the message should be sent to. Each group has a string name of no more then MAX_GROUP_NAME chars. The array should have at least as many group names as the 'num_groups' parameter indicates.

The Spread system will only send the message once but will deliver it to all connections which have joined at least one of the groups listed.

The *mess_type* is a short int (16 bits) which can be used by the application arbitrarily. The intent is that it could be used to NAME different kinds of data messages so they can be differentiated without looking into the body of the message. This value will be endian corrected before receiving.

If the non-scatter variants are being used, then a single buffer is passed to the multicast call specifying the full message to be sent. The *mess_len* field gives the length in bytes of the message. While the *mess* field is a pointer to the buffer containing the message. For a scatter call, both of these are replaced with one pointer, *scat_mess*, to a scatter structure, which is just like an iovec. This allows messages made up of several parts to be sent without an extra copy on systems that support scatter-gather.

RETURN VALUES

`NORMAL`	the number of bytes sent on success.
`ILLEGAL_SESSION`	the *mbox* given to multicast on was illegal.
`ILLEGAL_MESSAGE`	the message had an illegal structure, like a scatter not filled out correctly.
`CONNECTION_CLOSED`	during communication to send the message errors occured and the send could not be completed.

3.3.6 SP_receive and SP_scat_receive

sp.h

```
#include <sp.h>
int  SP_receive( mailbox mbox, service * service_type,
        char sender[MAX_GROUP_NAME], int max_groups,
        int * num_groups, char groups[][MAX_GROUP_NAME],
        int16 * mess_type, int * endian_mismatch, int max_mess_len,
        char * mess );
int  SP_scat_receive( mailbox mbox, service * service_type,
        char sender[MAX_GROUP_NAME], int max_groups,
        int * num_groups, char groups[][MAX_GROUP_NAME],
        int16 * mess_type, int * endian_mismatch, scatter * scat_mess );
```

`SP_receive` is the general purpose receive function for the Spread toolkit. This receives not only data messages, but also membership messages for the connection. Messages for all groups joined on this connection will arrive to the same mailbox, so a call to `SP_receive` will get a single 'message' from any one of the groups. After the receive completes, a number of fields are set to values indicating meta information about the message (such as groups, mess_type, endianness, type, etc).

This function is the most complex used in Spread because it is the only way for the system to return information to the application. The meaning of many of the fields changes depending on whether the message is a data message or a membership message.

The `SP_receive` function will block if no messages are available.

The *mbox* gives which connection to receive a message on. *Service_type* is a pointer to a variable of type 'service' which will be set to the message type of the message just received. This will be either a REG_MESSAGE or MEMBERSHIP_MESS, and the specific type.

The rest of the parameters differ in meaning depending on the *service_type*. If the *service_type* is a REG_MESSAGE (i.e. data message) then:

sender	a pointer to an array of characters of at least MAX_GROUP_NAME size. This will be set to the name of the sending connection(its private group name).
max_groups	the maximum number of groups you have allocated space for in the "groups" array passed to the receive function.
num_groups	a pointer to an int which will be set to the number of groups set in the "groups" array. See Section 3.1.1 for details on this field when the groups array is too small.
groups	array holds upto *max_groups* group names, each of which is a string of at most MAX_GROUP_NAME characters. All of the groups which are receiving this message will be listed

here, unless the array is too small and you have chosen DROP_RECV semantics by setting that flag in the *service_type* field when you called `SP_receive`. In that case as many group names as can fit will be listed and the *num_groups* value will be set to be negative. For example, if your groups array could store 5 group names, but a message for 7 groups arrived, the first five group names would appear in the *groups* array and *num_groups* would be set to 7.

`mess_type` set to the message type field the application sent with the original message, this is only a short int (16bits). This value is already endian corrected before the application receives it.

`endian_mismatch` set to true (1) if the endianness of the sending machine differs from that of this receiving machine. Otherwise set to false (0). This field is handled in a special way when certain errors are returned. See Section 3.1.1 for details on this field when the message buffers are too small.

`mess` the actual message body being received is stored into this buffer.

`max_mess_len` the length of the *mess* buffer in bytes. Messages larger then the buffer size are handled in the usual way. See Section 3.1.1 for details.

If the `SP_scat_receive` function is used instead of the `SP_receive` function then the *mess* and *max_mess_len* fields are replaced by a single *scat_mess* scatter structure. The scatter should be initialized to contain whatever buffers you wish to receive into and their lengths. These buffers must be valid memory areas. They will be filled in by the receive call in the order they are listed.

If this is a MEMB_MESSAGE (i.e. membership message) and it is specifically a TRANS_MESS type membership message, than:

`sender` set to the name of the group for which the membership change is occuring.

`max_groups` not used.

`max_mess_len` not used.

`num_groups` always set to 0.

`groups` is empty, since there are no normal groups for a transitional membership. The sender field is used instead.

`mess_type` set to -1.

`endian_mismatch` set to zero since the transitional does not have any endian issues.

mess	left empty.

So, in essence, the only information you get is the *sender* field which is set to the group name that received a transitional membership change message. The importance of the TRANS_MEMB_MESS is that it tells the application that all messages received after it and before the REG_MEMB_MESS for the same group are 'clean up' messages to put the messages in a consistant state before actually changing memberships. For more explanations of this please see other documentation and research papers.

If this is a MEMB_MESSAGE (i.e. membership message) and it is specifically a REG_MEMB_MESS type membership message, then:

sender	set to the name of the group for which the membership change is occuring.
max_groups	same as regular message.
max_mess_len	same as regular message.
mess_type	set to the index of this process in the array of group members.
endian_mismatch	set to 0 since there are no endian issues with regular memberships.
num_groups	set to the number of members in the group after the change.
groups	contains a deterministically ordered list of the private group names of the members of the group after the change.
mess	contains the identifier of this group membership and a list of all the private group names of those processes which came with your process from the old group membership into this new membership.

The data buffer will include the following fixed length fields:

- group_id;
- int num_members;
- char trans_members[][MAX_GROUP_NAME];

The groups array will have num_members group names, each of which is a fixed length string. The content of the groups array is dependent upon the type of the membership change:

CAUSED_BY_JOIN:	trans_members contains the private group of the joining process.
CAUSED_BY_LEAVE:	trans_members contains the private group of the leaving process.

`CAUSED_BY_DISCONNECT:` trans_members contains the private group of the disconnecting process.

`CAUSED_BY_NETWORK:` trans_members contains the group names of the members of the new membership who came with *me* (the current process) to the new membership. Of course, all *new* members can be determined by comparing it with the groups parameter of the `SP_receive` call.

If this is a MEMB_MESSAGE and it is *neither* a REG_MEMB_MESS nor a TRANS_MEMB_MESS, then it represents exactly the situtation where the member receiving this message has left a group and this is notification that the leave has occured, thus it is sometimes called a *self-leave* message. The simplest test for a self-leave message is if the message is CAUSED_BY_LEAVE and REG_MEMB_MESS is FALSE then it is a *self-leave* message. TRANS_MEMB_MESS never have a CAUSED_BY_ type because they only serve to signal upto where SAFE delivery and AGREED delivery (with no holes) is guaranteed in the complete *old* group membership.

The other members of the group this member just left will receive a normal TRANS_MEMB_MESS, REG_MEMB_MESS pair as described above showing the membership change.

The fields of `SP_receive` in the case of a self-leave will be as follows:

sender	set to the name of the group for which the membership change is occuring.
max_groups	same as for regular message.
max_mess_len	same as for regular message.
mess_type	set to 0.
endian_mismatch	set to 0.
num_groups	set to 0.
groups	will be empty. This is because this process is no longer part of the group and thus has no knowledge of it.
mess	contains the group_id of new membership and the private group name of the member who just left. This name should always be the private group name of the connection which received this message.

The data buffer will include the following fixed length fields:

- group_id;
- int num_members;

- char trans_members[][MAX_GROUP_NAME];

The trans_members array will have 1 group name containing the private group name of the leaving process, since this case only occurs with a CAUSED_BY_LEAVE membership change.

RETURN VALUES

NORMAL	Returns the size of the message received on success.
ILLEGAL_SESSION	the *mbox* given to receive on was illegal.
ILLEGAL_MESSAGE	the message had an illegal structure, like a scatter not filled out correctly.
CONNECTION_CLOSED	during communication to receive the message communication errors occured and the receive could not be completed.
BUFFER_TOO_SHORT	the message body buffer was too short to hold the message being received.
GROUPS_TOO_SHORT	the groups buffer was too short to hold the groups list or member list being received.

3.3.7 SP_equal_group_ids

sp.h

```
#include <sp.h>
int SP_equal_group_ids( group_id g1, group_id g2 );
```

SP_equal_group_ids provides a way to compare two group_id's that originated in membership messages. Since a group_id is considered an opaque type to the application programmer the only thing you can do with it is use it as an identifier for a membership view and compare it with other group_ids.

3.4 Miscellaneous Functions

Chapter 4

Spread Java API

4.1 Introduction

Writing[1] Spread applications in Java is as simple and easy as writing Spread applications in C, but with the added benefits of the Java language. All of the functionality of the C interface to Spread is available when developing in Java, with some extra tools and utilities. The Spread library consists of one package, ”spread”, which contains ten classes. The main classes are SpreadConnection, which represents a connection to a deamon, SpreadGroup which represents a spread group, and SpreadMessage, which represents a message that is either being sent or being received with spread.

The Spread package is contained in a file, ”spread.jar”. To use Spread from a Java application, this file should be in your classpath. For Java 1.1, this is done by making sure the directory containing spread.jar is in the CLASSPATH enviornment variable. For Java2 this is done by using the ”-classpath” option on the command line when compiling or running any classes that user Spread. For applets, simply put spread.jar in the same directory as the applet class. To access the Spread classes from any classes you write, simply include the following line at the top of the .java file:

```
import spread;
```

[1] A previous version of this chapter was written by Dan Schoenblum the original author of the Spread Java Library

4.2 API Datatypes

4.3 Spread Classes

4.3.1 SpreadConnection

import spread;
SpreadConnection SpreadConnection();
connect(InetAddress *spread_name*, int *port*, String *privateName*,
boolean *priority*, boolean *groupMembership*);
disconnect();
SpreadGroup getPrivateGroup();
multicast(SpreadMessage *message*);
multicast(SpreadMessage *messages[]*);
SpreadMessage receive();
SpreadMessage[] receive(int *numMessages*);
boolean poll();
add(BasicMessageListener *listener*);
add(AdvancedMessageListener *listener*);
remove(BasicMessageListener *listener*);
remove(AdvancedMessageListener *listener*);

To establish a connection to a spread daemon, use the SpreadConnection class. First, create a new SpreadConnection object, then use the connect() method to make a connection to a daemon:

```
SpreadConnection connection = new SpreadConnection();
connection.connect(InetAddress.getByName("daemon.address.com"), 0,
                   "privatename", false, false);
```

Figure 4.1: Establishing a Java connection to Spread

The first argument to connect() is an InetAddress, which is a class in the package java.net. The static method InetAddress.getByName() takes one argument, a String object specifying an Internet address, and returns an InetAddress object representing that address. The address can be passed either by name or by IP (A.B.C.D). Alternatively, if null is passed as the first argument to connect(), an attempt will be made to connect to a daemon on the localhost. The second argument to connect() is the port to connect to. If this is 0, the default port (4803) will be used.

The *private_name* is the name this connection would like to be known as. It must be unique on the machine running the spread daemon. The name can be of

at most MAX_PRIVATE_NAME characters with the same character restrictions as a group name (mainly it cannot contain the '#' character).

The *priority* is a 0/1 flag for whether this connection will be a "Priority" connection or not. Currently this has no effect.

The *group_membership* is a boolean integer. If 1 then the application will receive group membership messages for this connection, if 0 then the application will *not* receive *any* membership change messages.

This connection can be used until the disconnect() method is called, which terminates the connection to the daemon.

Aside from adding and removing listeners, no methods should be called on a SpreadConnection before connect() is called.

The *private_group* should be a pointer to a string big enough to hold at least MAX_GROUP_NAME characters. After the Connect call returns it will contain the private group name of this connection. This group name can be used to send unicast messages to this connection and no one can join this special group.

To receive a message, use SpreadConnection's receive() method. receive() will block until a message is available. When one is ready to be received, the message will be read and placed into a new SpreadMessage object which is returned by receive().

The isRegular() method can be used to check if the message is a regular message. Otherwise, it is a membership message. Membership messages will only be received if they are request by passing true as the final arguement to SpreadConnection's connect() method. If the message is a regular message, the get*() methods in SpreadMessage will provide more information about the message. If the message is a membership message, the getMembershipInfo() method can be used to return a MembershipInfo object, which provides information about the membership change.

```
if(message.isRegular() == true)
    System.out.println("New message from " + message.getSender());
else
    System.out.println("New membership message from "
                       + message.getMembershipInfo().getGroup());
```

Figure 4.2: Testing type of received Java Spread message

In addition to using SpreadConnection's receive() method, there is another way to receive messages. This is by the use of two interfaces: BasicMessageListener and AdvancedMessageListener. To use a listener, first implement one of these two interfaces. Then add them to a connection with one of SpreadConnection's add() methods:

After being added to a connection, the listener will be alerted whenever a new message is received on the connection. BasicMessageListener's have one

callback method, messageReceived(), which is called whenever a new message arrives. AdvancedMessageListener's have two callback methods: regularMessageReceived() is called whenever a regular message arrives, and membershipMessageReceived() is called when a membership message arrives. These methods will keep being called until the listener is removed from the connection with one of SpreadConnection's remove() methods:

There can be multiple listeners on a connection at any one time. SpreadConnection's receive() should not be called while the connection has any listeners.

4.3.2 SpreadMessage Class

```
import spread;
SpreadMessage SpreadMessage();
boolean isIncoming()
boolean isOutgoing()
int getServiceType();
boolean isRegular();
boolean isMembership();
boolean isUnreliable();
boolean isReliable();
isFifo();
isCausal();
isAgreed();
isSafe();
isSelfDiscard();
SpreadGroup[] getGroups();
SpreadGroup getSender();
byte[] getData();
Object getObject();
Vector getDigest();
short getType();
boolean getEndianMismatch();
setServiceType(int serviceType);
setUnreliable();
setReliable()
setFifo();
setCausal();
setAgreed();
setSafe();
setSelfDiscard(boolean selfDiscard);
addGroup(SpreadGroup group);
addGroup(String group);
addGroups(SpreadGroup groups[]);
addGroups(String groups[]);
setData(byte[] data);
setObject(Serializable object);
digest(Serializable object);
setType(short type);
MembershipInfo getMembershipInfo();
Object clone();
```

To multicast a message to one or more groups, use the SpreadMessage class.

First, create a new SpreadMessage object. This creates a new outgoing message. Next, the message data, the groups the message is going to, and the type of delivery requested should be set. This will use functions like `setData`, `addGroup`, and `setReliable`.

The setData() method sets the message's data to an array of bytes. Alternatives to setData() are setObject() and digest(), each of which takes an object that implements the Serializable interface. setObject() is used for sending one Java object, while repeatedly calling digest() can be used to send multiple objects in one message. The addGroup() method is used to specify a group to send the message to. The setReliable() is used to set the delivery method. Possible delivery methods are: unreliable, reliable, fifo, causal, agreed, and safe. The setDelfDiscard() method can be used to specify that this message should not be sent back to the user who is sending it.

To actually send the message, call SpreadConnection's multicast() method on the message you want to send.

4.3.3 SpreadGroup Class

SpreadGroup SpreadGroup();
join(SpreadConnection *connection*, String *groupname*);
leave();
String toString();
boolean equals(Object *object*);

To join a group on the connection, use the SpreadGroup class. First, create a new SpreadGroup object, then use the join() method to join a group:

```
SpreadGroup group = new SpreadGroup();
group.join(connection, "group");
```

Figure 4.3: Joining new group in Spread

The first argument to join() is the SpreadConnection on which the group is joined. This must be specified so that Spread knows which connection messages should be received on. The second argument is the name of the group to join.

Messages multicast to the group will be received on the connection until the leave() method is called.

4.3.4 MembershipInfo Class

```
boolean isRegularMembership();
boolean isTransition();
boolean isCausedByJoin();
boolean isCausedByLeave();
boolean isCausedByDisconnect();
boolean isCausedByNetwork();
boolean isSelfLeave();
SpreadGroup getGroup();
GroupID getGroupID();
SpreadGroup[] getMembers();
SpreadGroup getJoined();
SpreadGroup getLeft();
SpreadGroup getDisconnected();
SpreadGroup[] getStayed();
```

4.4 Factory Classes

4.4.1 MessageFactory

```
messageFactory = new MessageFactory(message);
messageFactory.setDefault(message);
SpreadMessage message = messageFactory.createMessage();
```

The MessageFactory class is a utility included with the Java interface to Spread. An object of the MessageFactory class is used to generate any number of outgoing messages based on a default message. To use a message factory, create a MessageFactory object, passing the default message to the constructor.

To change the default message at a later time, use the setDefault() method:

To get a message from the message factory, use the createMessage() method: The createMessage function will create a clone of the default message. Message factories with more complex behavior can be created by extending the MessageFactory class.

One example is a message factory that sets the message's data to the current system time:

4.5 Exceptions

When an error occurs in a Spread method, a SpreadException is thrown. One example is if receive() is called on a SpreadConnection() object before connect()

```
public class TimeStampMessageFactory extends MessageFactory
{
    public SpreadMessage createMessage()
    {
        SpreadMessage message = super.createMessage();
        message.setObject(new Long(System.currentTimeMillis()));
        return message;
    }
}
```

Figure 4.4: Creating a message factory for Java Spread

is called on that object. Another example is calling leave() on a SpreadGroup object before calling join() on that object. Any method that is declared as throwing a SpreadException must be placed within a try-catch block:

```
try
{
        connection.multicast(message);
}
catch(SpreadException e)
{
        e.printStackTrace();
        System.exit(1);
}
```

Figure 4.5: Testing errors on multicast in Java Spread

4.6 Notes for Applets

When using Java in an applet, there is usually a security manager installed, which restricts what your code is allowed to do. For example, when running an applet in a web browser, the code is not allowed to make Internet connections to any place other than the machine running the web server. So, if Spread is running in an applet, in a web browser, the spread daemon must also be running on the machine running the web server. The following code can be used, in the class that extends Applet, to get an InetAddress object for the machine running the web server:

```
InetAddress host = InetAddress.getByName(getCodeBase().getHost());
```

Figure 4.6: Getting Applet connection address

4.7 Miscellaneous Functions

Chapter 5

The Event Subsystem

5.1 Introduction

The Event subsystem in *Spread* provides an abstract interface to manage all possible types of events that can occur in a networked application. This includes network or file IO and timed function calls. These events are registered with subsystem along with the functions to be called when the events occur. The Event subsystem uses whatever tools the operating system provides to monitor system events and to wait for specified times to elapse to implement a main loop which calls the registered callback functions whenever appropriate.

A significant difference between the *Spread* event system and other similar wrappers around `select` or `poll` is that the event system also supports the idea of priority levels. Each event is registered at a particular priority level. At any time only events with a certain priority or higher will be handled. This feature is used in *Spread* to selectively ignore certain types of events (such as new client connections) while other more important events are going on (such as membership changes).

5.1.1 Initialization and General Use

Before using any of the Events functions, the system must be initialized by calling the `E_init` function. This allocates some data structures and initializes them to a correct starting state.

Once the system is initialized and any beginning file descriptors or queued functions are registered the main control loop can be started by using the `E_handle_events` function and can be exited by calling the `E_exit_events` function. It is possible for some small amount of additional work to be done after calling `E_exit_events` as it does not take effect until control reaches the main loop.

`events.h`

- `int E_init(void);`
- `void E_handle_events(void);`

- `void E_exit_events(void);`

5.2 Timekeeping Functions

The following functions should be used to avoid any system dependencies. These are ported to whatever native time interfaces each operating system provides. The `sp_time` structure is identical to a standard unix `struct timeval` and provides microsecond resolution to time.

```
typedef struct dummy_time {
        int32   sec;
        int32   usec;
} sp_time;
```

`events.h` The list of functions are given below. Their use is fairly obvious.

- `sp_time E_get_time(void);`
- `sp_time E_sub_time( sp_time t, sp_time delta_t );`
- `sp_time E_add_time( sp_time t, sp_time delta_t );`
- `int E_compare_time( sp_time t1, sp_time t2 );`
- `void E_delay( sp_time t );`

`E_compare_time` is defined to return 1 if $t1 > t2$, to return -1 if $t1 < t2$, and to return 0 if $t1 == t2$. `E_delay` acts like a unix `usleep()` function and will wait for the specified time (or more) before returning control back to the program.

5.3 Queued Events

One of the two main uses of the events system is to allow the application to call functions at a later time, or even to call them immediately after the current function completes. Since an application using the Events functions gives up its main control loop, in many cases a callback function will want to cause other functions to be executed soon after it completes, but it cannot call them itself[1].

`events.h` The two functions are:

- `int E_queue( void (* func)(), int code, void *data, sp_time delta_time );`
- `int E_dequeue( void (* func)(), int code, void *data );`

[1] This could occur either because doing so would cause other events to be delayed too long, or because the ordering of events in the functions requires one to compete before the other starts

The `code` and `data` parameters are passed to the function `func` when it is called at the specified time. The function can use them however it wants. In most cases you should use the `code` parameter if all you need to pass is an integer, for example, representing a file descriptor, a state value, or to distinguish between the normal case and a special case for the function. If you need to pass more complicated state, then create a structure which stores it all and pass a pointer to the structure in the `data` parameter.

Note that the `E_dequeue` function does not free any data pointed at by the `data` parameter so the application has to make sure to free that if noone besides this function call needs it.

5.4 Managing File Descriptors

Each file descriptor can be registered with a callback function for each type of event that can occur (reading, writing, and exceptions). When the file descriptor is ready for the requested action the callback function will be called. It will be passed the file descriptor in the first parameter, the `code` value in the second parameter and the `data` pointer in the third parameter. When a file descriptor is attached it is automatically set to the `active` state. However, if the priority threshold is not sufficient to include it, then it will be made inactive before the `E_attach_fd` function returns.

All of these functions return 0 on success and -1 on error.

The priority can be set to:

```
LOW_PRIORITY
MEDIUM_PRIORITY
HIGH_PRIORITY
```

The threshold for what priority of events should currently be handled is set by the `E_set_active_threshold` function which also returns what the threshold is set to. Note currently there is no way to query what the current threshold is without setting it. Without changing the threshold level, individual file descriptors[2] can be activated and deactivated by using the obvious functions. If a file descriptor is deactivated in this way, it will not be checked even if it is currently above the priority threshold or if a higher threshold is set later. It can only be activated by a call to `E_activate_fd` or by being detached and reattached. If a file descriptor is activated, it will *only* be actually checked if it is also above the current priority threshold. Thus, both a low threshold and a deactivate trump an activate or a high threshold.

One function can be registered for each (file descriptor, file event type) pair. For example one function can be registered for READ events on fd 5, another

[2]When I say “file descriptors” what I mean is a file descriptor and event type combination. For example, you can deactivate fd 5 for reading while keeping writing on fd 5 active

function for WRITE events on fd 5 and a third function for EXCEPTION events on fd 5. As a practical matter, you will usually want to register the same function for both READ and EXCEPTION events because the only way to detect when the other end of a TCP socket is closed is by doing a `read` or `recv` call on it and the return value being '0'. The closing of a TCP socket is sometimes considered a READ event by the operating system and sometimes an EXCEPTION event so registering both is necessary to correctly handle closed TCP sockets in all cases.

`events.h`

- `int E_attach_fd( int fd, int fd_type, void (* func)(), int code, void *data, int priority );`
- `int E_detach_fd( int fd, int fd_type );`
- `int E_activate_fd( int fd, int fd_type );`
- `int E_deactivate_fd(int fd, int fd_type );`
- `int E_set_active_threshold( int priority );`
- `int E_num_active( int priority );`