Service Sockets: Design, Implementation and Interfaces

Michael Schmitt, Anurag Acharya, Max Ibel
Dept of Computer Science
University of California, Santa Barbara

TRCS99-39
December 1999

Abstract

As network services have evolved, their communication structures have become increasingly complex. This increase in the complexity in the communication structure of services is mirrored by an increase in the complexity of the client code that interacts with and administers these services. In this paper, we propose the *Service Socket* abstraction. Conceptually, a Service Socket (or an S-socket) is a communication channel to a *service* instead of a particular *host*. Service Sockets abstract away the details of interacting with complex and dynamic services while retaining the conceptual simplicity, the familiar programming interface and the ease-of-use of Berkeley Sockets. The other attractive feature of the Service Sockets design is its modular construction: the service-specific components of a Service Sockets implement well-defined interfaces. Modularizing the different aspects of a service into a common core and pluggable components makes it easier to compose services as well as to reuse individual components. We describe the design and implementation of Service Sockets. Our experience with Service Sockets has been positive. The Service Sockets core consists of a shared library with a small footprint (60-75 KB). We have developed S-sockets for a wide variety of services. A microbenchmark-based evaluation indicates that the modular design of Service Sockets does not impose a major performance penalty.

1 Introduction

Since its introduction in the 4BSD releases, the socket abstraction has been the primary way in which applications interact with the network. Its wide acceptance results from a simple conceptual model and an easy-to-use operational interface. Sockets provide the notion of communication endpoints that can send and/or receive data. By abstracting the details of actual data transport from the sender to the receiver, sockets allow application writers to focus on *what data is to be communicated* instead of *how it is to be communicated*. Most of the first generation network services, such as telnet, ftp, finger, and mail delivery focused on data transport. The transport-level functionality provided by sockets was ideal for these services and most services were built around relatively simple communication structures – look up the IP address for the peer host, establish a socket (or a set of sockets) to the peer, and exchange data over the socket(s).

As network services have evolved, their communication structures have become increasingly complex. To improve service performance and reliability, for example, services are being replicated, requiring the clients to design and deploy some algorithm that selects the service replica to communicate with [3, 7, 10, 18]. To provide scalable feedback for real-time data delivery to large groups, RTP-based services require individual clients to keep track of loss characteristics and participate in a protocol (RTCP) that partitions the bandwidth allocated for feedback traffic among all receivers [24]. To handle group communication among peers with varying network connectivity, researchers have proposed several techniques (e.g., layered multicast [16], simulcast [13], and self-organized transcoding [13]) that send data over a set of related multicast groups and require clients to monitor loss characteristics on individual groups, join/leave groups based on loss characteristics and possibly perform transcoding for other members in the group. To provide scalable reliable data transfer to a large receiver group, techniques have been proposed that require clients to multicast their retransmission requests and to participate in a random-backoff-based protocol to avoid synchronizing retransmission requests [4, 8]. To provide uninterrupted service to mobile hosts traveling between zones served by different radio transmitters, a hand-off notification is sent to the client which then terminates its current service instance and requests service in the new zone.

This increase in the complexity in the communication structure of services is mirrored by an increase in the complexity of the client code that interacts with and administers these services. As the above examples show, instead of creating and using a (usually) small and (relatively) static set of sockets to exchange data with peers, clients for contemporary services have to manage, monitor and process a dynamically changing group of data transport channels, as well as cooperate with other clients and negotiate/adapt to changes in content resolution on individual channels, etc. Often, the code required for these operations is hand-crafted for each client application requiring the developers of each application to solve these problems anew. In some cases, application-class specific libraries have been developed. For example, the SmartSockets library from Talarian [25] for publish-subscribe applications, the Elemedia RTP library from Lucent [14] for applications requiring real-time data delivery and the ACE communication library from Washington University [23] for event-driven servers. These libraries are focused towards their application-class and define potentially complex and usually incompatible interfaces.

In this paper, we propose the *Service Socket* abstraction. Conceptually, a Service Socket (or an S-socket) is a communication channel to a *service* instead of a particular *host*. Operationally, an S-socket is a "higher-level socket" with a programming interface very similar to that of conventional Unix sockets. Just as conventional sockets hide the details of dealing with complex transport-level issues, S-sockets abstract away the details of interacting with complex and dynamic services. Retaining the conceptual simplicity, the familiar programming interface and the ease-of-use of the Berkeley Sockets interface was an explicit goal of our research. Figure 1 presents the Service Sockets API. We note that the interface parallels the Berkeley Sockets API except for the `socketSpec` parameter to `Ssocket()` that specifies the configuration of S-socket being requested.

We first present an overview of Service Sockets – the conceptual model, the programming interface and the design. In Section 3, we describe the implementation. Currently, full versions run on Solaris 2.5/6/7 and Linux 2.x; a scaled down version runs on Windows CE.[1] Service Sockets is implemented as a shared library with a small footprint (about 75 KB on Solaris and about 60 KB on Linux). A performance evaluation is presented in Section 4. In Section 5, we describe a number of specific S-sockets we have developed. These include a family of HTTP anycasting S-sockets which select the "best" member of a group of replicated HTTP-based servers; a layered multicast S-socket [16] to handle group communication among peers with varying network connectivity; and a family of S-sockets that interface with the RENS framework for rapidly evolvable network services [1]. We have used these RENS-based S-sockets to interact with a suite of distributed

[1]Service Sockets is built on top of standard sockets and pthreads. We expect porting it to other Unix platforms will be easy, but we have not confirmed this yet.

```
int Ssocket(char *socketSpec, int domain, int type, int protocol);
int Sconnect(int socket, const struct sockaddr *address, size_t address_len);
int Slisten(int socket, int backlog);
int Sbind(int socket, const struct sockaddr *address, size_t address_len);
int Sclose(int socket);
int Spoll(struct pollfd fds[], nfds_t nfds, int timeout);
int Sselect(int nfds, fd_set *readfds, fd_set *writefds, fd_set *errorfds,
            struct timeval *timeout);
ssize_t Ssend(int socket, const void *buffer, size_t  length, int flags);
int Ssendto(int s, const char *msg, int len, int flags,
            const struct sockaddr *to, int tolen);
int Ssendmsg(int s, const struct msghdr *msg, int flags);
ssize_t Srecv(int socket, void *buffer, size_t length, int flags);
int Srecvfrom(int s, char *buf, int len, int flags,
              struct sockaddr *from, int *fromlen);
int Srecvmsg(int s, struct msghdr *msg, int flags);
```

Figure 1: The Service Sockets API. Note that the interface parallels the Berkeley Sockets interface except for an additional parameter to `Ssocket()` that specifies the configuration of Service Socket being requested.

services including a location-dependent information service, customcast (customized multicast)[2], selectcast (selective multicast)[3], hierarchical chat servers, a hierarchical streaming audio network and caching proxies for intermittently connected devices. In section 6, we present a discussion of the Service Socket approach and related work. Finally, we present our concluding remarks in Section 7.

2 Overview

To drive the design of Service Sockets, we studied a large suite of network services including services that have been deployed as well as services that have been proposed in the literature. Our goal in this study was to categorize the functionality required by the client code that interacts with these services. We found that the client functionality for network services can be grouped into five categories:

- **Service setup:** includes determining how to connect to the service requested by the application, negotiating the service type, service protocol, content format, and resource requirements with the service provider as well as setting up the transport channels required for the service, etc. Examples of service setup functionality include: application-level anycasting that selects a member of a replicated server group [3, 7, 10, 18], looking up the SDR announcements for MBONE sessions, looking up the bandwidth information for layered multicast/simulcast sessions, and using a service location protocol to locate a service provider (e.g., SLP [26]), etc.

[2]Customcast is a filtering news ticker that adaptively distributes customized news content over a set of multicast groups [1].

[3]Selectcast provides a lightweight one-time multicast of messages to a sender-selected receiver group [1].

- **Data processing:** includes transducing the data format, resolution and communication protocols, application-level message fragmentation and reassembly, computing loss characteristics, and pacing the rate at which data is sent. Examples of data processing include encryption, compression, combining packets received on different groups in a layered multicast session [16], TCP-friendly rate-adjustment for UDP-based protocols [15, 20, 21], and protocol transduction for interfacing to heterogenous peers, etc.

- **Fault detection:** service-independent fault detection requires monitoring the state of transport-level channels used to interact with the service. The monitored state consists of the connection status for TCP-based channels, and heartbeat activity for UDP-based channels. Service-specific fault detection includes loss detection through sequence numbering, etc.

- **Fault recovery:** includes re-establishing one or more transport-level connections, finding a new service provider or a new service replica and re-initializing the service state, and re-negotiating the service type, service protocol, content format, resource requirements with the service provider.

- **Service control:** involves control-level interaction with the service provider (or peers) to smoothly administer the service. Examples of service control include: join/leave messages from peers on a multicast group, hand-off notification from the service provider when a mobile client is moving from the zone served by one radio transmitter to another, notification of the creation of new multicast group(s) carrying transcoded content [13], notification from a customcast service informing the client about changes in the multicast groups it should listen to [1], and quality-of-service feedback (e.g., RTP-based services [24]) retransmission requests for scalable reliable multicast [4, 8], etc.

We based our design of Service Sockets on this categorization. Our design consists of a *core* module, a *control layer* and any number of *data layers*. The *core* module is service-independent and is provided as a part of the Service Sockets infrastructure. The *control layer* and the *data layers* are service-dependent components that are written by service designers and are dynamically loaded as necessary. The Service Sockets design specifies a standard interface for each component. This allows service writers to easily integrate components.

The Service Sockets *core* provides the glue between the different layers in an S-socket as well as the functionality common to all S-sockets (e.g., loading and instantiating the service-specific layers, monitoring the status of the transport-level channels, initiating fault recovery if any of them has failed, and checking for control messages from the service provider). The *core* also handles merging/replicating the data from/to a set of transport-level connections. It maintains the global state for the S-socket and provides a well-defined callback interface to view/modify this state. This helps maintain an isolation between the different components of an S-socket and lets individual components evolve independently.

A *control layer* can: determine how to connect to the service requested by the application, negotiate the service type and resource requirements with the service provider, set up the transport channels required for the service, manage/monitor these transport channels, provide feedback about the service status/quality and react to dynamic changes in the service state. A *control layer* consists of one or more of the following components: *service setup*, *service control* and *fault recovery*. A *data layer* processes data that is exchanged between the client and the service. This can include transducing the data format, resolution and communication protocols, application-level message fragmentation and reassembly, computing loss characteristics, pacing the rate at which data is sent

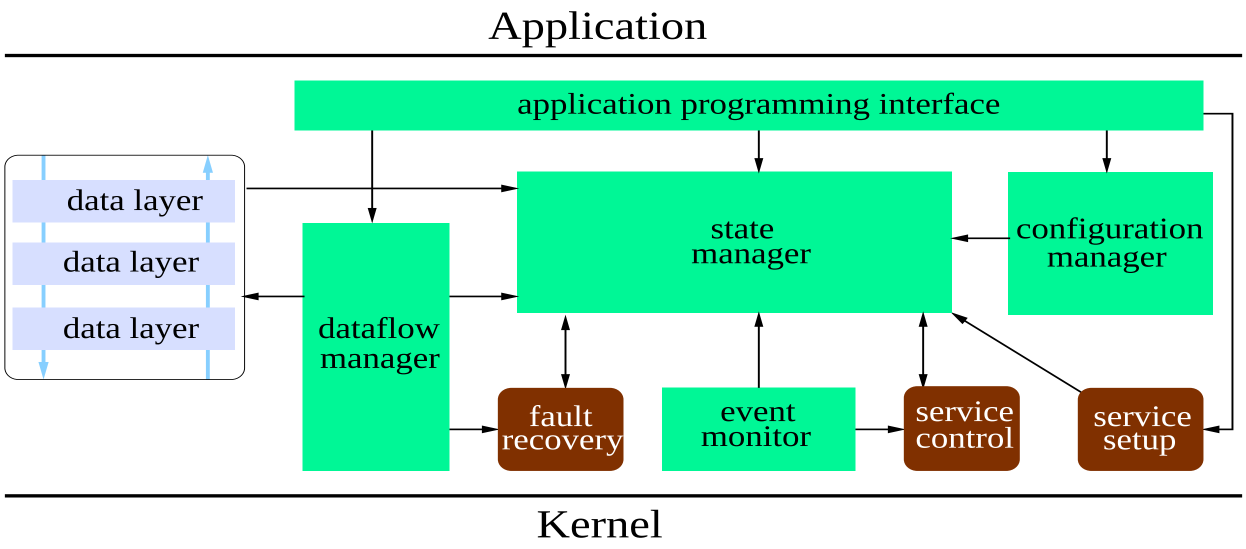


Figure 2: Architecture of an S-socket. Data layers (light blue) and control components (dark brown) can be plugged in and interact with the core (green).

etc. All Service Socket data layers must implement the same interface – this allows data layers to be stacked without being aware of each other.

3 Implementation

In this Section, we describe our implementation of Service Sockets. Figure 2 presents an architectural overview. The *core* module consists of five functional components: the *application programming interface (API)*, the *configuration manager*, the *dataflow manager*, the *event monitor* and the *state manager*. The *core* also provides default implementations for the components of the *control layer*.

The *core* module is organized around the *state manager* which encapsulates the S-socket state and provides `set` and `get` operations to manipulate this state. It checks changes to status-related fields of the state and triggers response routines if these fields are set to error values.

The *API* component coordinates the operations performed to implement the Service Socket API. To implement the `Ssocket()` call, it invokes the *configuration manager* to load, initialize and configure the dynamic libraries needed for this S-socket. It also creates a new mutex and associates it with this S-socket. This mutex is used by the *API* component to ensure thread-safety of subsequent operations on this S-socket. To implement the `Sconnect()` call to connect to the desired service, the *API* component invokes the *service setup* component.[4] To implement the `Ssend()` and `/Srecv()` calls, the *API* component invokes the *dataflow manager* to coordinate the data transfer. Note that every data transfer is explicitely initiated by the application; data is processed only when the application calls `Srecv()` or `Srecv()`.

The *dataflow manager* coordinates the data movement to and from the network and up and down the stack of *data layers*. If it detects a fault on one or more of the transport-level channels used (e.g., if a TCP connection is terminated), it invokes the *fault recovery* component. Note that this fault detection technique is service-independent but is applicable only to connection-oriented

[4]The *service setup* component is also invoked to implement `Sbind()` for multicast channels.

channels. In addition, *data layers* can report service-specific faults (e.g., poor quality observed) by setting the `faultOccured` variable in the S-socket state. The *state manager* checks changes to this variable. When this variable is set, the *state manager* triggers the *fault recovery* component.

The *event monitor* detects asynchronous events including arrival of control messages from the service provider or peer clients and timer expirations. The control messages are used to smoothly administer the service (for examples, see Section 2); the timer events are used to trigger periodic updates in soft-state-based services as well as to schedule periodic feedback on service quality (e.g. the RTCP receiver reports). These events can have an immediate impact on the service configuration and it is important to handle them even if the application makes no calls into the S-sockets library. To allow asynchronous detection and handling of these events, we create one "control" thread per S-socket. If this thread detects an event, it calls the *service control* component of the control layer to handle it. Note that this "control" thread is created only if a *service control* component has been specified or a timer event has been requested by the *service setup* component.

In the rest of this section, we provide more details about the *configuration manager*, the *state manager*, the *event monitor*, and the *dataflow manager*. We also describe the requirements Service Sockets imposes on the *control layer* components that are plugged in.

3.1 The Configuration Manager

The configuration manager is invoked by the *API* component inside the call to `Ssocket`. It parses the `socketSpec` argument to `Ssocket` which specifies the configuration for the S-socket being created. This argument is an ASCII string that consists of a list of component specifications separated by `|` characters (this is the sheffer stroke or the Unix pipe symbol). Each component specification consists of a component name with an (optional) list of arguments which are specified as parameter-value pairs. For instance, a Service Socket that compresses and then encrypts (decrypts and decompresses) may be created by calling:

```
Ssocket("compr_uncompr|crypt_decrypt(key='q>Bw-KS',seed='42')",AF_UNIX,SOCK_STREAM,0).
```

The configuration manager parses this specification from left to right and creates the stack of components for the S-socket. Components that occur earlier in the specification are positioned higher in the stack. The configuration manager looks up each component in the `.ssocketrc` file.[5] The `.ssocketrc` file contains a sequence of component descriptions. See Figure 3 for a sample configuration file. Each component description consists of a name and an address. The address is specified as `<layer>://<location>/<parameters>/` where `layer` field is one of `data`, `control`, or `macro`, indicating that the component is a data layer, a control layer, or a macro, respectively. The `location` field specifies the dynamically linked library (DLL) that implements the component and the `parameters` field specifies parameters for the component. On encountering a data or a control component, the configuration manager appends it to the bottom of the stack it is constructing.

A macro component is a composition mechanism that allows service designers to use a single name to refer to pre-created S-socket configurations. For example, in the sample configuration file in Figure 3, the `lmc` component is a pre-created configuration for layered multicast. It consists of two layers, the `lmc_data` data layer and the `lmc_control` control layer. Libraries that implement macro components provide an `expand` routine which returns an ASCII string with the same format as `socketSpec`. On encountering a macro component, the configuration manager immediately loads

[5]It looks for this file in the directories listed in the SSOCKET_PATH environment variable. If this variable is not defined, it looks in the user's home directory.

```
# ---------------------------------------------- macro definitions
lmc             macro://lmc.so/
# ------------------------------------------- data layer definitions
compr_uncompr   data://compr_uncompr.so/
lmc_data        data://lmc.so/
# ---------------------------------------- control layer definitions
rens_forward    control://rens.so/lookup=foo.cs.ucsb.edu:7890/
lmc_control     control://lmc.so/
```

Figure 3: Sample configuration file.

the corresponding DLL and calls its `expand` routine. It then replaces the macro component in the `socketSpec` by the string returned by `expand` and the parsing continues.

Once the parsing is completed, the configuration manager checks the component stack for correctness: the DLLs for all components must exist, at most one *control layer* must be present and it must be the last layer in the list. It then loads all the DLLs and initializes them by calling the `init` routine.[6] Finally, the configuration manager initializes the S-socket state, including filling a function table with the entry points of the different components.

3.2 The State Manager

Successful modularization of different aspects of a service into separate, reusable components requires that individual components interact only through well-defined interfaces. To this end, the state manager encapsulates all the global state of an S-socket and provides a `set/get` interface to manipulate it. A second advantage of using such an interface is that it allows the state manager to check changes to specific fields of the state and to trigger response routines if these fields are set to error values. This allows any control/data layer component to report the occurrence of service-specific faults and to initiate fault recovery.

The rest of this subsection describes the individual parts of the S-socket state and the interface provided to manipulate each part.[7]

File descriptors: an S-socket has one or more transport-level channels, each with its own file descriptor. The set of file descriptors is initially created by the *service setup* component and can be modified by the *fault recovery* component or the *service control* component. The state manager provides a `get_fd/set_fd/insert_fd/delete_fd` interface to set and retrieve file descriptors and the associated information. File descriptors can be labeled as data descriptors or control descriptors.[8] An S-socket can have at most one control descriptor and an arbitrary number of data descriptors.[9] If it exists, the control descriptor is monitored by the *event monitor* for the arrival of control messages from the service provider. The data descriptors are used by the *dataflow manager* for network I/O.

Timers: each file descriptor can have a timer associated with it. Timers can be associated with

[6]The list of parameters specified in the address of a component (in `.ssocketrc`) are passed to its `init` function.
[7]For the complete interface for the *core* module (including the state manager), please see Appendix A.
[8]Since labeling is implemented as a bit field, service-specific labeling can be used to further distinguish descriptors.
[9]Nothing in the design of Service Sockets prevents from using multiple control descriptors. However, none of the services we considered or implemented would benefit from multiple control descriptors within the same S-socket.

multiple file descriptors. They can be used to send heartbeats each time the timer expires, as well as to trigger fault recovery if no heartbeat is received within a given period. Timers can be manipulated using a `set_timer/get_timer/delete_timer` interface.

Blackboard: the state manager provides a "blackboard" that can be used by different service-specific components to communicate in a clean way. A blackboard is a global data structure that all components can write to or read from [6]. It allows components to store state between invocations. It also allows components to communicate without having to know about each other. The blackboard contains attribute-value pairs – attributes are arbitrary ASCII strings and values are simply `void*` pointers. The state manager provides a `set_attr/get_attr/delete_attr` interface to access the blackboard. See section 5 for examples of the use of the blackboard.

Status variables: the state manager provides two status variables which it monitors: `faultOccurred` and `envChange`. When the `faultOccurred` variable is set, the state manager invokes the *fault recovery* component; when the `envChange` variable is set, the *service control* component is invoked. A component that sets one of these variables posts additional information regarding the fault or the environment change to the blackboard by convention.

Mutexes: the state manager associates two mutexes with every S-socket. One of these is used by the API component to ensure thread-safety; the other is used to synchronize network access between the application thread(s) and the control thread.

3.3 The Event Monitor

The *event monitor* detects asynchronous events including arrival of control messages from the service provider and timer expirations. To allow asynchronous detection and handling of these events, we create one thread per S-socket if the `service setup` has created a control descriptor or timers. The *event monitor* repeatedly calls `poll` on the control channel for the S-socket with the timeout value for `poll` set to the first pending timer expiration. Whenever an event is detected (the `poll` times out or there is a pollable event on the control channel), the event monitor calls the *service control* component. It passes a description of the event including the channel descriptor/timer that caused the event. After the return from the *service control* component, the monitor updates the set of timers, recalculates the next timer expiration and continues with the `poll` loop.

3.4 The Dataflow Manager

The *dataflow manager* coordinates all data movement in an S-socket: to and from the network and up and down the stack of *data layers*. Every *data layer* has an `send` routine and a `recv` routine. Both these routines take a vector of data buffers as input and return a vector of data buffers as output.[10] When receiving data (for `Srecv`, `Srecvfrom` or `Srecvmsg`), the *dataflow manager* first reads data from the transport-level channels at the bottom of the *data layer* stack. It then proceeds up the stack. At each layer, it calls the `recv` routine with the data returned by `recv` routine in the previous layer. A `recv` routine is allowed to buffer data as it deems fit. This allows *data layers* to perform application-level message fragmentation and reassembly.

If this upward sweep through the stack results in more data being generated than requested by the application, the excess data is buffered by the Service Socket library. If on the other hand, less data than desired is generated, the behavior depends on whether the `MSG_WAITALL` socket option is set. If this option is not set and at least one byte of data is available, the available data is returned

[10] Please see Appendix A for the complete *data layer* interface.

to the application with a suitable return value. If the `MSG_WAITALL` option is set[11] or if zero bytes of data are available at the top of the processing stack, the upward sweep operation is repeated until enough data is available.

When sending data (for `Ssend`, `Ssendto` or `Ssendmsg`), the *dataflow manager* performs a downward sweep through the stack. At each layer, it calls the `send` routine with the data returned by `send` routine in the previous layer. A `send` routine is allowed to buffer data as it deems fit. This allows it to implement application-level framing [5] as well as rate control. All the data available at the bottom of the stack is sent to the network.

The *dataflow manager* can detect service-independent fault events as it accesses the network. It calls the *fault recovery* component whenever the *send/recv* operations fail for a transport-level channel. After successful fault recovery, it performs an upward sweep through the stack. At each layer, it invokes the *reset* routine to perform cleanup operations.

The synchronization between the *dataflow manager* and the *event monitor* poses a challenge. The *dataflow manager* accesses the data channels of an S-socket in the context of the application thread(s). The *event monitor* accesses the control channel of an S-socket in the context of its control thread. Both of these can invoke the *fault recovery* and the *service control* components. As a part of their operation, these components can change the set of channels for the S-socket. Since these threads run asynchronously, this can result in a race condition between the two contexts where one tries to change the set of channels and the other tries to read/write from/to the set of channels. We use a mutex-protected flag in the S-socket state to avoid this race condition. The mutex is locked when either of the contexts calls a *control layer* component (*fault recovery* or *service control*). The flag is set if the component modifies the set of channels associated with the S-socket. The mutex is released when the *control layer* component completes. Both the *dataflow manager* and the *event monitor* check this flag after each network operation. If the flag is found to be set, both reset their state and repeat the operation with the new set of channels.

3.5 Pre and post-conditions for control-layer components

The control layer components, *fault recovery*, *service control* and *service setup* perform control operations that change the state of the S-socket and thereby govern the operation of the *core* module. To ensure some measure of consistency in the interactions between these components and the *core* module, we have augmented the interface of these components with a set of pre- and post-conditions. When a component is called, the *core* guarantees that the corresponding pre-condition holds; when it returns, it checks to make sure the corresponding post-condition holds. If a post-condition check fails for any component, the *core* marks the S-socket as irrevocably failed. If this happens, all subsequent operations on this S-socket except `Sclose()` fail. The current set of pre- and post-conditions is:

Service setup: As a pre-condition, the S-socket is guaranteed to have no open data or control channels. As post-conditions, all status variables for the S-socket must be clear and at least one data channel must exist. If a control channel or a timer is present, the *service control* component must be enabled.

Service control: As a pre-condition, the `envChange` status variable is set. All post-conditions are

[11] The *dataflow manager* cannot pass a MSG_WAITALL flag to `recv/recvfrom/recvmsg` since it does not know beforehand how the *data layers* will shrink or expand the received data. For instance, if the application wants to read 10KB from a compress/uncompress S-socket, there might be only 1KB of (compressed) data ready for reading from the network.

the same as for *service setup*.

Fault recovery: As a pre-condition, the `faultOccured` status variable is set. All post-conditions are the same as for *service setup*.

We realize that these are rather weak post-conditions. Ideally, we would like to require (and enforce) more stringent post-conditions. For example, if the *fault recovery* component is invoked in response to a failed data channel, we would want to ensure that when this component completes, all data channels are fault-free. Attractive as this sounds, enforcing this is hard. The control component's code may close and reopen arbitrary transport-level channels. Or worse still, it could enter an infinite loop and never terminate. While this sounds rather drastic, we would like to point out that we run much the same risk when we call any library routine whose source code we have not personally inspected. In the current version of Service Sockets, we rely on the cooperation of service designers.

4 Performance and Overhead

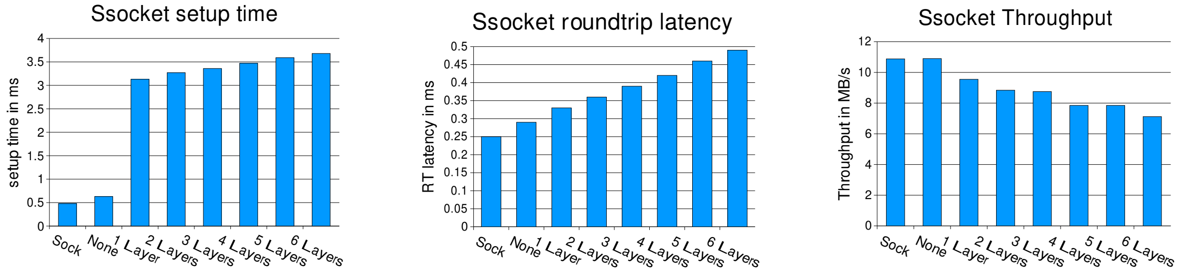
The main goal of Service Sockets is to abstract away the complexity of interacting with complex services. While performance is not an explicit goal, it is important to ensure that the Service Socket implementation does not incur an unacceptable overhead. To estimate this overhead, we performed a set of microbenchmark-based experiments that measured the setup time, end-to-end latency and throughput for Service Sockets. These experiments emulate the worst case. In real-life services, like the S-sockets described in Section 5, the service-specific code in data and control layers or long-distance networks connections may cause significant extra delays and marginalize the overhead inherent in S-sockets.

For these experiments, we compared the performance of three configurations of a microbenchmark. It consists of two peers that establish a connection between themselves and exchange messages in a ping-pong fashion. The first configuration uses standard sockets to connect the peers, the second configuration uses S-sockets without any *data layers*, and the third configuration uses S-sockets with a varying number of a simple *data layer* that copies the data as it forwards it. In the third configuration, both peers use the same number of *data layers*. We used three metrics: the time for connection setup, the roundtrip latency for a 1-byte message, and sustained bandwidth for a long transfer. All experiments were repeated until the measurements established a 90% confidence interval within 5% of the mean. These experiments were conducted on a cluster of SUN Ultra-30 machines connected by a fast Ethernet switch.

Figure 4 presents these metrics (connection setup time, roundtrip latency, throughput) for all three configurations. We note that for all three metrics, performance of S-sockets with no *data layers* is close to that of standard sockets (within 5% for throughput, and within 50 μsec for latency, and within 150 μsec for connection setup). The connection setup time for S-sockets increases sharply (by about 2500 μsec) when the first *data layer* is added and slowly thereafter. By inserting timing probes, we determined that almost all the additional cost for adding a *data layer* to an S-socket is due to the dynamic linking operation.

The roundtrip latency increases linearly with the number of *data layers*. Each additional layer adds 30 μsec to the roundtrip latency – which is the cost of invoking each *data layer* in the stack. The throughput decreases roughly linearly with the number of *data layers* – this is due to the copying overhead in forwarding layer.

From these results we conclude that the modular design of Service Sockets is unlikely to impose a major performance penalty.



(a) connection setup (b) roundtrip latency (c) throughput

Figure 4: Overhead introduced by Service Sockets. "Sock" stands for standard sockets, "None" stands for S-sockets with no *data layers*, "n Layers" stands for S-sockets with n *data layers*.

5 Applications

We have used the Service Sockets framework to develop S-sockets for a variety of services including a family of HTTP anycasting S-sockets which select a server from a group of replicated HTTP-based servers; a layered multicast S-socket [16] to handle group communication among peers with varying network connectivity; and a family of S-sockets that interface with the RENS framework for rapidly evolvable network services [1].

5.1 Family of HTTP Anycasting S-sockets

Anycasting allows a client to select a server for a replicated service. An anycast-based service consists of two operations: finding the set of replicated servers that provide the service and selecting a server from a group of replicas. We consider three ways to perform each operation:

- **Finding the set of servers:** the first scheme uses `gethostbyname()` which can return multiple IP addresses for the same domain name. This is often the case for servers that use round-robin DNS for load-balancing [12]. In this scheme, the local DNS resolver provides caching and consistency. The second scheme requires replicated services to place the list of alternatives in a well-known location on each server. We propose that replicated HTTP servers store the list of mirrors in the `/mirrors.txt` file. This file contains a line-delimited list of entries. Each entry contains the URL of a replicated service followed by a list of mirror URLs.[12] The third scheme uses a general-purpose mirror service that takes the original service URL as an argument and returns a set of servers that provide that service. The last two schemes can optionally provide a TTL for mirroring information that can be cached on disk by the anycasting code as appropriate.

- **Selecting a server:** the first scheme is rough and simple. It uses the top-level domain of the server domain names to filter out servers that are *known* to be far away. For this purpose, it considers `.com`, `.edu`, `.mil`, `.net` and `.org` as equivalent. The second scheme uses the time to make a TCP connection as the performance metric. It selects the server that takes the least amount of time to create a connection. The third scheme computes a more complex metric: the time to set up a TCP connection, make a well-known and valid request (`GET /`) and receive the reply.

[12] We base this design on the "/robots.txt"-based scheme used by HTTP servers to control the operation of web crawlers.

Based on these techniques, we have developed a family of HTTP anycasting S-sockets that can be used to automatically select a server for a replicated HTTP-based service. These S-sockets cover the HTTP anycasting parameter space as described above (three ways to find the servers and three ways to select a server from a group). The service URL is passed by the application in the `socketSpec`. The anycasting functionality has been implemented as a part of the *service setup* component of the S-sockets *control layer*. Note that this is a generic component that imposes no restriction on the *data layers* it can be composed with.

Service setup: this component takes the service URL parameter and parses it to extract the desired information. If required, it also maintains a disk-based cache of the replica information. After finding the set of servers and selecting a server (by using techniques described above), it makes the TCP connection, sends the HTTP request and notifies the state manager about the new stream (using the `set_fd` callback).

5.2 A Layered Multicast S-socket

Layered multicast is a technique to deliver multimedia content to a heterogenous group of receivers. In layered multicast, a media stream is encoded as a series of layers l_1, l_2,...,l_n such that a decoder with an ordered subsequence of the layers (i.e., l_1, l_2,...,l_i where $1 \leq i \leq n$) can usefully decode that subsequence even if the other layers are absent [16]. The lowest layer is called the base layer and the higher layers are called enhancement layers. The service provider sends each layer on a different multicast group. Each receiver can individually tailor the service to its network connectivity by selecting the subsequence of layers. The monitored packet loss rate on the different multicast groups can be used to determine the desired layers and the corresponding multicast groups to join.

In our layered multicast setup, the server partitions the data stream on each multicast group into a sequence of *data frames*. Each data frame includes a timestamp which is used by clients to synchronize the combination with corresponding data frames from the other layers. Our layered multicast S-socket consists of the *control layer* and one *data layer*. The *control layer* for this S-socket contains the *service setup* and the *fault recovery* components.

Service setup: this component takes two arguments: the domain name of the layered multicast server and the session identifier for the desired multimedia session. It contacts the server using a TCP connection and requests information about the configuration of layered multicast for the desired multicast session (number of multicast groups used, address and the expected bandwidth requirement for each group). Since it has no information about the available network bandwidth, it joins all the multicast groups for the sessions (changes to the set of multicast groups are done by the fault recovery component). It then notifies the *core* module about these multicast channels (using `set_fd`). It also saves the information about the configuration of the layered multicast for the just established session on the blackboard.

Data layer: the *dataflow manager* receives from all the multicast channels and presents a sequence of buffers to the *data layer* (each buffer corresponds to data from a different group). The data layer maintains a sliding window based on the timestamps. Each slot has an associated timestamp. A slot in the window is released if and only if all previous slots have been released and all the data frames for the current timestamp have been received. In this case, the data frames with this timestamp are combined and the result is passed up to the application. If the window fills up, the data layer drops data frames from incomplete slots at the head of window (i.e., the ones that should have been received the earliest). When the last data frame for a slot is dropped, the slot is deleted from the window. If a data frame arrives too late (i.e., after the corresponding slot has been deleted), it is dropped.

The data layer keeps track of the total data-loss rate as a fraction of data frames lost over t seconds, where t is a parameter. If the data-loss rate is greater than a high-water mark over a t-second interval or is lower than a low-water mark over a $2t$-second interval, it notifies the *state manager* about a fault condition by setting `faultOccured`. In both cases, it saves the current data-loss rate on the blackboard.

Fault recovery: this component manages all changes in the set of multicast groups subscribed to. Fault recovery is initiated by the state manager after faults have been indicated by the *data layer*. The *fault recovery* component retrieves the current data-loss rate and the configuration of layered multicast for the current session (number of multicast groups used, address and the expected bandwidth requirement for each group) from the blackboard. Depending on the high- and low-water marks and the data-loss rate, the number of multicast groups being received is incremented or decremented, and the S-socket state is updated with the new set of received groups.

5.3 Family of RENS-based S-sockets

As a part of the RENS project, we are currently developing a framework for rapidly evolvable network services [1]. RENS services consist of distributed components connected by named communication channels. The communication structure of RENS services is specified explicitly, and components are programmed in an event-driven manner, having only knowledge of local state. The RENS runtime environment leverages its knowledge of the communication structure to provide automatic instantiation of distributed services, detection of and transparent recovery from common faults, and scaling on demand. This allows service designers to focus on service-specific code. The component-based structure of RENS services also makes it easier to extend, reuse or modify these services. We have developed a diverse suite of RENS services including location-dependent information servers, customcast (customized multicast) which is a filtering news ticker that adaptively distributes customized news content over a set of multicast group [1], selectcast (selective multicast) which provides a lightweight one-time multicast of messages to a sender-selected receiver-group [1], hierarchical chat servers, a hierarchical streaming audio network and caching proxies for intermittently connected devices.

RENS services can interact with legacy applications. However, to gain the maximum benefit from the automatic deployment, fault-recovery and scaling characteristics of RENS, clients need to participate in the RENS protocols for service setup and fault-recovery. To facilitate this, we have developed a family of RENS-based S-sockets. The functionality to interact with RENS has been implemented as the *service setup*, *fault-recovery* and *service control* components. Note that these components are generic and impose no restrictions on the *data layers* they can be composed with. As a result, it is relatively easy to construct S-sockets for a variety of RENS services.

Service setup: three steps are needed to setup a RENS service. The first step is to acquire a description of the service. A RENS service description consists of the components of the service and the communication channels between these components. It is specified as a graph (using the domain-specific language WRENCH) with components as the nodes and communication channels as the edges. Desired properties of the components and channels are specified as node and edge attributes. The second step is to find a server that can host the service and to send the description to this server. The third step is to receive a description of the deployed service including the locations of all the components and the port numbers used for all the channels.

To acquire the WRENCH description for a service, *service setup* can either contact a directory service[13] or use locally cached descriptions. It parses the WRENCH description and extracts from it the location of the server that will host the first component.[14] It also updates the service description to specify a control port that can be used by RENS to communicate control information back to the client. It connects to the server and sends the service description. If no acknowledgment arrives on the control port before a timeout occurs, the setup is aborted. The acknowledgment indicates that the service deployment is complete and includes information about the deployed service including the locations of all the components and the port numbers used for all the channels. This component saves both the WRENCH service description, the deployment information, and the created control and data file descriptors on the blackboard.

Service control: the control channel created by the *service setup* component is monitored by the *event monitor*. This channel is used by the components of the service to provide feedback to the client as well as to notify it about changes in its environment. Typical control messages include location updates (when service components have been relocated due to faults), termination announcements (preparing for an intentional termination) and hand-off notifications (if the set of channels between the client and the service changes in any way). The event monitor invokes the *service control* component whenever a message is received on the control channel. This component retrieves the deployment information from the S-socket state and updates it as necessary. In case of a hand-off, it closes/creates new channels as needed and updates all the changed information in the S-socket state.

Fault recovery: this component is invoked either when a TCP-based channel is prematurely terminated, or when the `faultOccurred` status variable for the S-socket is set. In case of a terminated connection, it retrieves the deployment information from the blackboard and tries to re-establish the connection. If this fails (within a timeout), it assumes the service component has failed and tries to re-instantiate the failed component at the same location. If this fails, it retrieves the original service description from the blackboard, determines a new location for the component (possibly using a directory lookup) and tries to instantiate it there. If this fails too, it gives up and terminates the S-socket. See [1] for more details on the fault-recovery algorithm.

6 Discussion and related work

The most attractive aspect of the Service Sockets library is that it abstracts away the details of interacting with complex and dynamic services while retaining the conceptual simplicity, the familiar programming interface and the ease-of-use of Berkeley Sockets. This approach of providing a common and easy-to-use interface to a diverse collection of resources has had a long and successful history in operating systems. Prominent examples include the use of the *file-system* interface to manipulate and interact with disk-resident files, devices, processes (`/proc` in Plan 9/Solaris/Linux/Irix), protocols (`/dev/{tcp,udp}` in Solaris) and the use of the *socket* interface to interact with a variety of network protocols (*/usr/include/sys/socket.h* lists twenty-five different protocols). Using a common interface allows programmers to operate on a wide variety of resources without having to learn and use a large suite of potentially incompatible interfaces.

Previous efforts at abstracting the details of interacting with services have been limited to specific application-classes. Examples include the SmartSockets library from Talarian [25] for

[13] Currently, we use an extensible version of OpenLDAP [19] that allows the directory to invoke user-specified functions as a part of the lookup operation [1].

[14] Occasionally, this can require another lookup operation.

publish-subscribe applications, the Elemedia RTP library from Lucent [14] for applications requiring real-time data delivery and the ACE communication library from Washington University [23] for event-driven servers. Such libraries are focused towards their application class and provide idiosyncratic and usually incompatible interfaces.

Another approach to abstracting the details of interacting with services is taken by object-oriented techniques like CORBA [9], DCOM [2] and Jini [28]. These techniques automate service lookup, service setup and data transfer by using fixed service-independent techniques. They do not allow service designers to customize these operations. Furthermore, they are more or less limited to request-response interactions. It is hard to fit services that stream data to the clients (e.g., layered multicast) in these approaches.

The other attractive feature of the Service Sockets design is its modular construction: the service-specific components of Service Sockets implement well-defined interfaces and interact only with the *core* and that too through a narrow interface. Modularizing the different aspects of a service into components makes it easier to compose services as well as to reuse individual components. In advocating modular construction of code that interacts with the network, we follow the lead of several successful efforts including STREAMS [22], x-kernel [11] and Scout [17].

6.1 Potential limitations

Attractive as the Service Sockets approach appears, it has a few potential limitations. First, the current implementation of Service Sockets handles incoming data only when the application transfers control into the Service Sockets library. In some cases (e.g., large unexpected bursts of incoming data arriving while the application is busy doing something else), this can lead to buffer overflow and packet drop. While we have not seen this phenomenon, an application that anticipates an occurrence can prepare for it by having a dedicated thread loop over a call to `Srecv()`.

Second, it is currently not possible to use the standard `poll()` and `select()` calls for a mix of standard file descriptors and S-socket descriptors. This limitation arises from the fact that our implementation of Service Sockets is user-level. Note that this does not limit application functionality. The `Sselect()` and the `Spoll()` routines provided by the Service Sockets API support both kinds of descriptors.

Third, the Service Sockets library currently requires cooperation of the plugged-in components. It may be possible to use software fault isolation (SFI) [27] to ensure that the components manipulate the internal S-socket state in a consistent manner. We have not yet investigated how well SFI works for this purpose.

Finally, the current design of Service Sockets does not support (de)multiplexing data between multiple Service Sockets and a single transport-level channel. It also does not allow state sharing between multiple S-sockets. These restrictions significantly simplify the Service Sockets design and increase its robustness. On the other hand, this restriction means it is not possible, for example, to implement S-sockets that multiplex multiple short-lived TCP connections over a single TCP connection or that distribute data from a single feed to multiple applications. Note, however, that it is easy to get around these restrictions by using a separate process for the multiplexing/demultiplexing. Therefore, we believe that our design choice is a good trade-off between generality and simplicity.[15]

[15] To paraphrase Einstein, an abstraction should be as general as necessary and no more general :)

7 Concluding remarks

In this paper, we have proposed the *Service Socket* abstraction. Conceptually, a Service Socket is a communication channel to a *service* instead of a particular *host*. The primary advantage of the Service Sockets approach is that it allows application programmers to use complex and dynamic services without having to learn and use a large suite of potentially incompatible interfaces. Furthermore, programmers can leverage their familiarity with the Berkeley Sockets interface.

The other attractive feature of Service Sockets is the modular design: the service-specific components of an S-socket implement well-defined interfaces and interact only with the *core*, using a compact and convenient interface. Modularizing the different aspects of a service into components allows for easy composition and reuse of service components. For example, the anycasting components described in Section 5.1 can be reused for a wide variety of HTTP-based services. The careful modular design approach has also allowed us to achieve a small program text size (60-75 KB) without explicit effort in this regard.

Our experience with Service Sockets has been positive. We have been able to develop Service Sockets for a wide variety of services including a family of HTTP anycasting S-sockets, a layered multicast S-socket, encryption and compression S-sockets and a family of S-sockets that interact with the RENS framework for rapidly evolvable network services. A microbenchmark-based evaluation indicates that the modular design of Service Sockets does not impose a major performance penalty. In the worst case, a simple S-socket incurs less than 5% overhead for throughput, less than 50 μsec overhead for roundtrip latency and less than 150 μsec overhead for setup when compared to a conventional socket.

We are currently exploring ways to port Service Sockets to mobile clients and devices. A scaled down version of Service Sockets runs on Windows CE, and we are investigating other platforms. Mobile clients are relatively resource-limited and operate in environments with dynamically changing resource availability. It is, therefore, important for mobile clients to be able to dynamically adapt to changes in their operating environment. We believe that the Service Sockets framework can be used to integrate a variety of adaptation techniques while retaining a simple and familiar interface.

Acknolwedgements

We would like to thank Mathias Kölsch, Guido Brückman and Robert Chalmers for their feedback on previous versions of this paper.

References

[1] A. Acharya, M. Ibel, M. Schmitt, M. Kölsch, D. Magdic, B. Smith, and M. Tuncer. Wrens: A Framework for Rapidly Evolvable Network Services. Technical Report TRCS99-31, Dept of Computer Science, University of California, Santa Barbara, Sep 1999.

[2] N. Brown and C. Kindel. Distributed component object model protocol – DCOM/1.0[16]. 1998.

[3] R. Carter and M. Crovella. Dynamic server selection using bandwidth probing in wide-area networks. Technical Report BU-CS-96-007, Dept of Computer Science, Boston University, 1996.

[16] Available at http://msdn.microsoft.com/library/specs/distributedcomponentobjectmodelprotocoldcom10.htm

[4] Y. Chawathe, S. Fink, S. McCanne, and E. Brewer. A proxy architecture for reliable multicast in heterogeneous environments. In *Proc. of ACM Multimedia'98*, 1998.

[5] D. Clark and D. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proc. of SIGCOMM'90.*

[6] R. Engelmore and A. Morgan. *Blackboard Systems.* Addison Wesley, 1988.

[7] Z. Fei, S. Bhattacharjee, E. Zegura, and M. Ammar. A novel server selection technique for improving the response time of a replicated service. In *Proc. of Infocom'98.*

[8] S. Floyd, V. Jacobson, C. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. In *Proc. of SIGCOMM'95*, pages 342–56.

[9] O. M. Group. *CORBA 2.2/IIOP Specification.* http://www.omg.org/corba/c2indx.html, 1997. OMG formal document 97-09-01.

[10] J. Guyton and M. Schwartz. Locating nearby copies of replicated internet servers. In *Proc. of SIGCOMM'95*, pages 288–98.

[11] N. Hutchinson and L. Peterson. x-kernel: an architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, January 1991.

[12] E. Katz, M. Butler, and R. McGrath. A scalable HTTP server: The NCSA prototype. *Computer Networks and ISDN Systems*, 27:155–64, 1994.

[13] I. Kouvelas, V. Hardman, and J. Crowcroft. Network adaptive continuous-media applications through self-organised transcoding. In *Proc. of NOSSDAV'98.*

[14] Lucent Technologies. The Elemedia RTP library. *Available by ftp on request*, 1999.

[15] J. Mahdavi and S. Floyd. TCP-friendly unicast rate-based flow control. Technical note sent to the end2end-interest mailing list[17], Jan 1997.

[16] S. McCanne, V. Jacobson, and M. Vetterli. Receiver-driven layered multicast. In *Proc. of SIGCOMM'96*, pages 117–30, 1996.

[17] D. Mosberger and L. Peterson. Making paths explicit in the Scout operating system. In *Proc. of OSDI'96*, 1996.

[18] A. Myers, P. Dinda, and H. Zhang. Performance characteristics of mirror servers on the internet. In *Proc. of Infocom'99.*

[19] Open source for LDAP software and information. *http://www.openldap.org.*

[20] J. Padhye, J. Kurose, D. Towsley, and R. Koodli. A TCP-Friendly Rate Adjustment Protocol for Continuous Media Flows over Best Effort Networks. Technical Report TR 98-04, Dept of Computer Science, University of Massachusetts, Amherst, Oct 1998.

[21] R. Rejaie, M. Handley, and D. Estrin. RAP: An end-to-end rate-based congestion control mechanism for realtime streams in the Internet. In *Proc. of IEEE INFOCOM'99.*

[17] Available at *http://www.psc.edu/networking/papers/tcp_friendly.html*

[22] D. Ritchie. A stream input/output system. *AT&T Bell Labs Technical Journal*, 63(8 Part 2):1897–1910, Oct 1984.

[23] D. Schmidt. The ADAPTIVE communication environment[18], 1994.

[24] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. Network Working Group RFC 1889, Jan 1996.

[25] Talarian: Our Products: SmartSockets[19], 1999.

[26] J. Veizades, E. Guttman, C. Perkins, and S. Kaplan. Service location protocol. Network Working Group RFC 2165, Jun 1997.

[27] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Proc. of SOSP'93*, pages 203–16.

[28] J. Waldo. Jini architecture overview. *http://www.sun.com/jini/whitepapers/architecture.pdf*, 1999.

A Component Interfaces

In this section, we describe the interfaces defined by the Service Sockets design for different Service Sockets components. We present four interfaces: the *data layer* interface which specifies the signature that must be implemented by each *data layer* component, the *control layer* interface which specifies the signature that must be implemented by the different components of a *control layer*, the *macro* interface which specifies the signature that must be implemented by a component that implements a Service Sockets configuration macro and the *callback* interface which specifies the signature of the callback routines provided by the *core* module. The callback routines are used by *data layer* and *control layer* components to interact with the Service Sockets *core*. A macro component is a composition mechanism that allows service designers to use a single name to refer to pre-created S-socket configurations.

A.1 The Data Layer Interface

Figure 5 presents the routines that must be implemented by a *data layer* component and their signatures. The first two parameters for all routines in the *data layer* interface are identical: an S-socket identifier `sid` and a component instance identifier `cid`. The component instance identifier is a unique identifier (unique for all instances of all components of all S-sockets being used by an application) that is assigned by the *core*. It is used by *data layer* components as a key to store and retrieve component-instance-specific data in the blackboard maintained by the *core*. Assigning a unique instance identifier in this manner allows multiple instances of a component to efficiently save and retrieve instance-specific state.

dl_init(): this routine is called by the *configuration manager* as a part of configuring a component. The third parameter for this routine is a list of parameters for this data layer from the `socketSpec`.

[18] *available at http://www.cs.wustl.edu/ schmidt/papers/sug-94.ps.gz*
[19] *available at http://www.talarian.com/products/smartsockets/smartsockets.shtml*

`int dl_init(int sid, int cid, par_t *pars)`
`buf_t* dl_recv(int sid, int cid, buf_t *buf)`
`buf_t* dl_send(int sid, int cid, buf_t *buf)`
`int dl_close(int sid, int cid)`
`int dl_reset(int sid, int cid)`

Figure 5: The data layer interface

It is expected to allocate and initialize the state for the component and save it on the blackboard. It should return zero on success and non-zero on failure.

dl_send(): this routine is called by the *dataflow_manager* as a part of implementing an `Ssend`, an `Ssendto` or an `Ssendmsg` call. The third parameter for this routine is a list of buffers – either directly from the application or from the output of the data layer immediately above this one in the data layer stack. Each buffer can be labeled/tagged to identify its contents. It is expected to process, modify, fragment or buffer the data. It returns a list of buffers. If the list of buffers is empty (i.e., NULL), the *dataflow manager* terminates its downward sweep through the data stack.

dl_recv(): this routine is called by the *dataflow_manager* as a part of implementing an `Srecv`, an `Srecvfrom` or an `Srecvmsg` call. The third parameter for this routine is a list of buffers – either directly from the network or from the output of the data layer immediately below this one in the data layer stack. Each buffer can be labeled/tagged to identify its contents. It is expected to process, modify, reassemble or buffer the data. It returns a list of buffers. If the list of buffers is empty (i.e., NULL), the *dataflow manager* terminates its upward sweep through the data stack. If after an upward sweep is terminated, less data than desired is available at the top of the data layer stack, the upward sweep resumes at the bottom of the stack and is repeated until enough data is available.

dl_close(): this routine is called by the *configuration manager* as a part of shutting down an S-socket. It is expected to free the space used by a component. It should return zero on success and non-zero on failure.

dl_reset(): this routine is intended for by the *fault recovery* component - which is expected to call `dl_reset()` for every component in the data layer stack.[20] This routine is expected to perform a rollback to a consistent state – ready to restart operation after recovering from a fault. It should return zero for success and non-zero for failure.

The *data layer* components are not supposed to access the network directly since they are only used for streaming data through a stack. This means these components should not create/manipulate sockets. Currently, we assume a cooperative environment and do not enforce this restriction.

A.2 The Control Layer Interface

Figure 6 presents the signatures that must be implemented by the different control layer components, if present. The first parameter for all routines in the *control layer* interface is an S-socket

[20] Note that since the *fault recovery* component is user-specified and is a pluggable component, there is no guarantee that this routine is called.

`int service_setup(int sid, serv_t *service)`
`int service_control(int sid, int label)`
`int fault_recovery(int sid, int label)`

Figure 6: The control layer interface

identifier `sid`. Note that, in contrast with *data layer* components, *control layer* components do not have a component identifier. This is because each S-socket has at most one instance of a *control layer* component (e.g., the anycasting *service setup* component) but can (possibly) have multiple instances of a *data layer* component.

Note that all three components (*service setup*, *service control* and *fault recovery*) need not be present in a *control layer*. The *core* uses a default functionality if a particular component is not present, e.g. if a fault occurs and a `fault_recovery` component is not present, the S-socket is marked irretrievably faulty and every call to the Service Sockets API except `Sclose` returns error.

service_setup(): this routine is called by the API component of the *core*. The second parameter is a list of parameters for the *control layer* from the `socketSpec` and the layer definition in `.ssocketrc`. It is expected to perform the operations necessary to find a service-provider, negotiate service parameters and establish connection to the service. It also creates and initializes the the state for the *control layer*. It should return zero on success and non-zero on failure.

service_control(): this routine is called by the *event monitor* component of the *core* when it detects an event of some sort. Events can be of two kinds: arrival of control messages from the service provider or peer clients and timer expirations. The second parameter consists of a label which identifies the file-descriptor(s)/timer(s) associated with the event. It is expected to perform the necessary operations for processing these events. For example, control messages often are used to smoothly administer the service; the timer events are used to trigger periodic updates in soft-state-based services as well as to schedule periodic feedback on service quality (e.g. the RTCP receiver reports). It should return zero on success and non-zero on failure.

fault_recovery(): this routine is called from two call-sites: by the *dataflow manager* when it detects that one or more of the transport-level connections used to interact with the service have terminated; by the *state manager* when it detects the `faultOccurred` flag to be set. It is expected to repair the faulty state or return an error. It is expected to return zero for success and non-zero for failure.

A.3 The Callback Interface

Figure 7 presents the signatures for the routines provided by the *core* for use by the pluggable components. The first parameter for all routines in the *control layer* interface is an S-socket identifier `sid`. Since the pluggable components for Service Sockets are implemented as dynamically linked libraries that are implemented separately, we need some way of providing/linking the callback routines for these components. Some platforms, such as Solaris, provide this functionality as part of the `dlopen` call used to load a dynamically linked library. Others, such as Linux, do not. To provide an efficient yet portable solution, we have developed a two-pronged technique. First, we provide an `.h` file that defines a `cb_init()` routine, a function vector and set of preprocessor macros to access this function vector. This `.h` file is to be `#include`d by each component. This results in a

`int set_labelbit(int sid, int label, char *name)`
`int clear_labelbit(int sid, int label, char *name)`
`fd_t* get_fd(int sid, int label)`
`int set_fd(int sid, int fd, int lis_fd, int label)`
`int delete_fd(int sid, int label)`
`tim_t* get_timer(int sid, int label)`
`int set_timer(int sid, int period, int label )`
`int delete_timer(int sid, int label)`
`void* get_data(int sid, char *name)`
`int set_data(int sid, char *name, void *data)`
`int set_flag(int sid, int flag)`
`int clear_flag(int sid, int flag)`

Figure 7: The callback interface

`cb_init()` routine and a global variable representing the callback function vector to be created in each DLL. As a part of Service Sockets configuration, the *configuration manager* creates a function vector containing pointers to all the callback routines. It then calls the `cb_init` routine for each DLL loaded. This routine saves the function vector argument in the global function vector variable. The callback routines can then be invoked using by indexing in this vector. The preprocessor macros hide the details of the function vector and the indexing operations.

Labels are implemented as bit fields. The LSB defines if this is a control or a data file descriptor. All other bits can be set/cleared in a service-specific manager through the `set/clear_labelbit` calls which provide a registry for used/unused bits in the labels of an S-socket. Each used bit is associated with a symbolic (that is, string) `name`. If a `name` has not yet been used, an unused bit is allocated for the operation and `name` is associated with it. The return value of `set/clear_label` differs from the `label` argument by having the bit associated with `name` set/cleared (if it isn't set/cleared already in `label`). Note that this mechanism used to dynamically allocate bits in a label is somewhat similar to the mechanism used to accommodate X Window System extensions by dynamically allocating event identifiers for extensions.

get/set/delete_fd: these routines operate on the descriptors for the transport-level connections used for an S-socket. The `set_fd` call associates a `label` with a descriptor `fd`. Accepting sockets can, optionally, also set the `lis_fd` argument. The `get/delete_fd` calls return/delete descriptors that (string-) match their `label` argument. These routines are for use by *control layer* components. The `set/delete_fd` calls return 0 if successful, non-zero otherwise; the `get_fd` call returns a descriptor if successful, NULL otherwise.

get/set/delete_timer: these routines operate on timers for an S-socket. They are similar in spirit to the `get/set/delete_fd` routines. A timer can be associated with one or descriptors. The `set_timer` call defines the length of the timer using the `period` parameter (in μsec).

get/set_attr: these routines operate on the blackboard for an S-socket. They use a symbolic name to identify the blackboard entry they are working on. The entry itself is passed and retrieved as a `void*` pointer.

set/clear_flag: these routines operate on the flags of an S-socket. Currently, Service Sockets sup-

```
char *expand(int sid, char *name)
```

Figure 8: The macro interface

ports only two flags: `faultOccured` and `envChange` which are used to request a call to *fault_recovery* and *service_control*.

A.4 The Macro Interface

Figure 8 presents the signature that must be implemented by all macro components. As mentioned earlier, a macro component is a composition mechanism that allows service designers to use a single name to refer to pre-created S-socket configurations. A macro component must implement the `expand` routine which maps the name used to refer to the configuration by the configuration itself. On encountering a macro component, the configuration manager immediately loads the corresponding DLL and calls its `expand` routine. It then replaces the macro component in the `socketSpec` by the string returned by `expand` and the parsing continues.