Service Sockets: A Uniform User-Level Interface for Networking Applications

Michael Schmitt, Anurag Acharya, Max Ibel, Costin Iancu
University of California, Santa Barbara

Abstract

As network services have evolved, their communication structures have become increasingly complex. This has led to a corresponding increase in complexity of applications that interact with these services. In this paper, we propose the *Service Socket* abstraction to help alleviate this complexity. Conceptually, a Service Socket represents a communication channel to a *service* instead of a particular *host*. The goal of Service Sockets is to allow application programmers to interact with diverse complex services using the familiar Berkeley Sockets programming interface. The Service Sockets framework provides service-independent functionality in a small core library and encapsulates service-dependent functionality in dynamically linked libraries with a well-defined interface. It provides a simple specification interface that can be used by application programmers to describe the desired configuration of a Service Socket. We present several examples of how Service Sockets can be used to provide a common interface for complex end-point functionality. A microbenchmark-based evaluation indicates that such an extensible user-level framework does not necessarily incur a substantial performance penalty.

1 Introduction

Since its introduction in the 4BSD releases, the socket abstraction has been the primary method by which applications have interacted with the network. Its wide acceptance resulted from a simple conceptual model and an intuitive interface: sockets provide the notion of communication end-points that can send and receive data. By abstracting the details of data transport, sockets allow application writers to focus on *what* is to be communicated instead of *how* it is to be communicated. The transport-level functionality provided by Berkeley Sockets was ideal for the first generation network services; they were built around a relatively simple communication behavior – look up the IP address for a peer, establish a connection (or a set of connections), and exchange data over the socket(s). As network services have evolved, their communication structures and their interactions with clients have become increasingly complex. For example:

- To improve service performance and reliability, services can be replicated. Clients of such services need to deploy a mechanism that selects a service replica [2, 20].
- To manage real-time data delivery to large groups, a scalable feedback mechanism can be deployed. Clients need to participate in a protocol to partition feedback bandwidth based on monitored traffic and loss characteristics (e.g., RTCP [26]).
- To establish group communication among peers with varying network connectivity, one of several techniques (e.g., Layered Multicast [17], Simulcast [14]) can be deployed to send data over a set of related multicast groups. Clients need to (de)multiplex and join or leave groups based on monitored loss characteristics and possibly perform content format conversion for other members in the group.
- To guarantee scalable reliable data transfer to large groups, some form of reliable multicast can be deployed [3, 8]. Clients need to multicast retransmission requests and participate in a random-backoff-based protocol to avoid synchronizing retransmission requests.

- To provide uninterrupted service to mobile hosts traveling between zones served by different radio transmitters, a hand-off protocol can be used. Clients need to react to hand-off notifications (or initiate hand-off themselves) by terminating the current service connection and re-requesting service in the new zone.

The above examples show that clients for contemporary services have to perform a variety of tasks: in addition to just transporting data, they have to manage, monitor and process a dynamically changing group of sockets, cooperate with other clients and adapt to changes in content, format or protocol on individual sockets, and so on. Often, the code for these operations is hand-crafted, requiring developers to solve the problems anew for each application. In some cases, application-class specific libraries have been developed. For instance, the SmartSockets library from Talarian [28] supports publish-subscribe applications, the Elemedia RTP library from Lucent [15] supports real-time data delivery and the ACE library from Washington University [25] supports event-driven communication. Since these libraries are focused towards their application class and define potentially complex and usually incompatible interfaces, users face a steep learning curve for each new library.

In this paper, we propose the *Service Sockets* 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" that allows application programmers to interact with diverse complex services using the familiar Berkeley Sockets programming interface. Just as Berkeley Sockets hide the details of dealing with complex transport-level issues, S-sockets hide the details of (and the code for) interacting with a variety of network services. Figure 1 presents the Service Sockets API. The only difference with the Berkeley Sockets API is an additional configuration parameter (*serviceSpec*) to the socket creation call (`Ssocket()`) which specifies the service-level functionality.

Main Contributions

- We introduce the Service Sockets abstraction that allows application programmers to interact with diverse complex services using the familiar Berkeley Sockets programming interface.
- We present our implementation of a Service Sockets framework. It provides service-independent functionality in a small core library (60-75KB depending on the platform) and places service-specific functionality in dynamically linked component libraries. Currently, our implementation runs on Solaris 2.5/6/7 and Linux 2.x; a scaled-down version runs on Windows CE.
- To demonstrate that extensibility and configurability of our user-level framework is achieved without a substantial performance price tag, we present measurements illustrating the overhead of the various parts of Service Sockets.
- We present a summary of our experience with building a diverse suite of S-sockets which includes a translating IRC S-socket, a layered multicast S-socket, and an HTTP anycasting S-socket.

2 Objectives and Requirements

The Berkeley Sockets framework has stood the test of time and has been successfully used to provide a uniform and conceptually simple programming interface for a wide variety of network transport protocols. Our goal for Service Sockets is to emulate this for diverse service-level (or application-level) protocols. To understand the requirements of contemporary and future network services, we studied a large suite of services – both existing and proposed. Our goal was to categorize the functionality required by the application code that interacts with these services.

Stackable Data Processing: Many network services require one or more forms of "data processing". This can include converting data formats, changing data resolution, adapting communication protocols, application-level message fragmentation and re-assembly, computing loss characteristics, and pacing the rate at which data is sent. Examples include encryption, compression, distillation [9], application-level framing [4], combining packets re-

```
int Ssocket(char *serviceSpec, int domain, int type, int protocol);
int Sconnect(int socket, const sockaddr_t *address, int address_len);
int Slisten(int socket, int backlog);
int Sbind(int socket, const sockaddr_t *address, int 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 *errfds,
            timeval_t *timeout);
int Ssend(int socket, const void *msg, int length, int flags);
int Ssendto(int socket, const void *msg, int len, int flags,
            const sockaddr_t *to, int tolen);
int Ssendmsg(int socket, const struct msghdr *msg, int flags);
int Srecv(int socket, void *msg, int length, int flags);
int Srecvfrom(int socket, void *msg, int len, int flags,
              sockaddr_t *from, int *fromlen);
int Srecvmsg(int socket, struct msghdr *msg, int flags);
```

Figure 1: The Service Sockets API.

ceived on different groups in a layered multicast session [17], TCP-friendly rate-adjustment for UDP-based protocols [16, 22, 23], protocol transduction for interfacing to heterogeneous peers, etc.

Some services require multiple levels of such processing. These can usually be organized as a sequence or a stack of modules that data is "streamed" through. This structure makes it easy to compose independently developed components and has been repeatedly and successfully used in the design of networking systems [24, 11, 19].

Event-Driven Control Handling: A large part of the increasing complexity of network services is due to an increase in the complexity of interactions each peer needs to participate in. This can include protocols to partition feedback bandwidth based on monitored traffic and loss characteristics (e.g., RTCP [26]), selecting a member of a replicated server group [2, 7, 10, 20], hand-off notification for mobile clients, sending retransmission requests for scalable reliable multicast [3, 8], dynamically varying the set of multicast groups listened to based on monitored loss characteristics [17], participating in a content transcoding protocol for other members in the group [14], etc.

These interactions are often asynchronous in that the application needs to rapidly recognize a situation and take appropriate action. As the number and the variety of situations that need to be dealt with increases, it becomes necessary to insert potentially complex checks at many places in the application. Control handling can usually be better dealt with by structuring it in an event-based manner with event detection and dispatch being performed by a common component and event handling being performed by service-specific event handlers. This allows the application to retain its natural organization and (in many cases) allows individual event handlers to be used for many services. Events can be external (peer closes a connection, communication is received on a control channel, etc) or internal (a timer expires, packet loss drops below a threshold, etc).

Configurability: To be able to use a uniform interface to interact with diverse services requires configurability. For ease-of-use, unprivileged applications should be able to specify service configurations without requiring an a priori setup by a privileged user. Furthermore, applications should be able to use multiple services during each invocation and different services during different invocations.

An increase in configuration options for a system usually also increases the set of errors that can occur. Since errors are relatively common in distributed/networking applications, it is important to provide application programmers with accurate error information. Since the same common interface is used for a wide variety of services, it is important to provide an extensible but common error reporting mechanism.

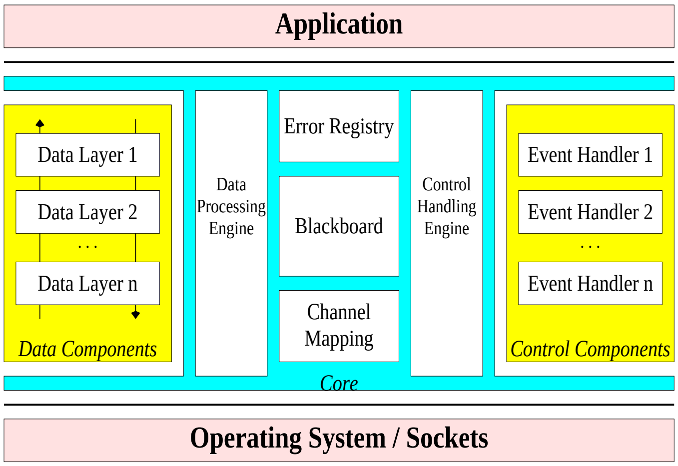

Figure 2: Design of the Service Sockets Architecture

3 Design and Implementation

Service Sockets is implemented as a user-level library interposing all network-related activity. It consists of a *core* library implementing functionality used across all services and service-specific *component* libraries that are loaded on-demand. Applications can employ multiple S-Sockets and configure each of them differently. A configuration consists of a list of component specifications and is passed to the library through a parameter of the `Ssocket()` call. Other than this extra parameter, the signature and semantics of the Service Sockets interface is the same as for Berkeley Sockets.

Figure 2 presents an overview of the Service Sockets architecture. The guiding principle of the design is to enable the modularization of services into control handling and data processing components. Any interaction between two components is mediated by the *core*. Components only interact with the core and are unaware of the functionality of other components.

Control components focus on handling specific control events while *data components* (or *data layers*) focus on specific data processing tasks. To handle the interaction with control components, the core provides support for event detection and dispatching through a *control handling engine*. To handle the interaction between data layers, the core provides a *data processing engine* that performs data movement up and down the stack of data layers.

These components are generally not stateless. State may have to be maintained between different invocations of one component (e.g., for application-level framing) or even be shared between different components (e.g., between RTP and RTCP components). The core provides the *blackboard* for this purpose by allowing to store and retrieve key-value tuples.

Complex network services may require clients to use multiple transport-level channels, i.e., sockets. These can be abstracted to one service-level channel, i.e., an S-Socket. While the Service Sockets architecture – as a whole – shields an application from the transport level, the *channel mapping* in the core shields the components of an S-Socket from the underlying transport channel(s). We allow two kinds of channels: *data channels* which are administered by the data processing engine and *control channels* which are administered by the control handling engine.

Any design of an extensible system needs to pay special attention to the consistent handling of component-specific error conditions. The core provides an *error registry* that can be used to record component-specific errors and to propagate them to the application.

The core invokes components through well-defined entry points. Each component has to provide entry points for initialization and shutdown. Data layers have to provide additional entry points for receiving data to push up or down the data layer stack. Control components can register additional entry points as specific control event handlers. All components can "callback" into the core to access the blackboard, the channel mapping and the error registry.

In the following, we explain the individual concepts in more detail. We describe how application programmers can specify the configuration of an S-socket; how components are managed through the data processing and control handling engines; and how the blackboard, the channel mapping and the error registry works.

Configuration: A service specification is passed as the first parameter (*serviceSpec*) to the `Ssocket()` call (see Figure 1). The core parses the specification, determines the requested service-specific component(s), loads the component libraries (DLLs), initializes them and organizes them into the desired configuration.

serviceSpec is a string consisting of a list of component specifications. The mapping between the names of components, the types of components (data, control or macro[1]) and the location of the dynamic libraries implementing them is stored in `.ssocketrc`, the Service Sockets configuration file. Component-specific parameters can be assigned as name–value pairs both through the *serviceSpec* and the configuration file.

Data components that occur earlier in the specification are positioned higher in the data layer stack. Since we impose no ordering on control event handlers, the order of control components in the specification is not important. S-socket configuration concludes with storing component-specific parameters from both *serviceSpec* and `.ssocketrc` – if any – on the blackboard, and calling the initialization entry points of each component. A sample *serviceSpec* and `.ssocketrc` for the IRC S-socket described in Section 5.1 is shown in Figure 4.

Data Processing: The *data processing engine* coordinates the flow of data from the application through the data component/layer stack into the network and vice versa. Every data layer has a `send()` routine and a `recv()` routine taking input buffers as an argument and producing output buffers as a result. Each buffer corresponds to a transport-level channel used by the S-socket. If multiple channels are present, the data layers are supposed to (de)multiplex the channel buffers between the application buffer and the network.

When receiving data (in `Srecvfrom()`, `Srecv()` or `Srecvmsg()`), the core first reads data from the transport-level channels and passes it to the bottom-most data layer in the stack. Rising to the top of the stack, the core calls each layer's `recv()` routine with the data returned by the `recv()` routine in the previous layer. A data layer is allowed to buffer data on the blackboard as it deems fit, allowing for application-level message fragmentation and reassembly. After the top-most data layer is processed, its resulting output buffer is then presented to the application. 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 core. Likewise, if less data than desired is generated, the core will either return the available data to the application or access the network and perform another upward sweep to gather more data, depending on whether the `MSG_WAITALL` flag is set. Transmissions to the network (in `Ssendto()`, `Ssend()` or `Ssendmsg()`) are handled symmetrically, i.e., the application buffer is passed down the data layer stack before sending the resulting buffer(s) on the network.

Control Handling: The *control handling engine* detects service-specific and service-independent events and dispatches them to corresponding event handlers which have been registered by control components. We use three kinds of events:

- *Network events* are generated when a control channel is ready or when any TCP channel fails. Network events are useful for services with dedicated data and control channels and for TCP-based fault tolerance.

- Components can flag certain state on the blackboard as a trigger. *Trigger events* are generated whenever a component writes to a trigger. Trigger events are useful for services which share transport channels for both for data and control. Triggers can be used to dispatch data processing and control handling to the right components.

- *Timer events* are generated when a timer expires. Components can register timers with the control handling engine. Timer events are useful for services that require periodic control handling (e.g., for gathering of session statistics) or for fault tolerance (e.g., to implement heartbeats).

Both control channels and timers are inherently asynchronous. We use a dedicated *monitor thread* to detect asynchronous events. When a synchronous or asynchronous event is detected, the control handling engine invokes the event handler registered for that

[1]The specification can be simplified by using *macro* components. A macro is a "high-level" service name that can expand to component lists. This can be used to hide the internal details of a pre-configured service.

event. It passes information about the event (e.g., the connection that failed, the timer that expired or the value that was assigned to the trigger) as a parameter. In addition, an event is generated by the core whenever `Sconnect()` or `Sbind()` are called by the application. This allows to perform service-specific lookup, connection and negotiation.

The control handling engine can execute both in the context of the monitor thread (e.g., timer expires, control channel ready) and in the context of application thread(s) (e.g., write to trigger, TCP data channel failure). The invoked event handlers in either context may modify the sets of control and data channels (e.g, for fault recovery).

Blackboard: A blackboard is a global data structure that all components can write to or read from [5]. The Service Sockets core provides a blackboard to maintain service-specific state between different invocations of the same component and to share service-specific state between different components. It contains key-value pairs – keys are arbitrary ASCII strings and values point to arbitrary data. Key-value pairs have to be explicitly flagged if they are to be shared. The key name for state that is not flagged as shared has to be unique only in the context of the component that uses it. Key names for shared state has to be unique for all components of the S-socket. Only shared state can be additionally labeled as a trigger.

Components that want to share state have to agree on a key name and on which component initially registers the key-value pair on the blackboard. Name clashes can be detected by distinguishing "post" operations (fail when name is already present) and "update" operations (fail when name is not present) when components write to the blackboard. If two components try to "post" shared state under the same key name, the second "post" will fail.

Channel Mapping: The channel mapping maintains the set of transport-level channels – i.e., sockets – currently used by the S-socket. Channels can be registered/removed by event handlers, (e.g., those for service setup or fault recovery). After an event handler has created or closed a channel to be used by the S-socket, it needs to register this with the core. Channels can be labeled as *data* or *control* channels. Data channels are used for communicating data to/from peers and are accessed by the data processing engine on explicit application requests (e.g., `Ssend()`, `Srecv()`, `Spoll()`, etc). Control channels are polled by the control handling engine asynchronously in the *monitor thread*. If a service requires channels which transport both data and control information, we can use the trigger mechanism to dispatch to the proper component.

Error Registry: Different components have different error conditions. These need to be managed in a global fashion to avoid conflicts between error codes. Components can register a symbolic error name and a corresponding error message with the error registry in the core. The registry assigns a unique error number with every symbolic error name. The component can subsequently indicate error conditions to the core by presenting its symbolic error name. The application can access the error message corresponding to the last set error number with `Sperror()`. This can be seen as an extensible user-level version of the standard BSD error reporting mechanism for system calls.

4 Performance and Overhead

It is important to ensure that a modular and extensible system like Service Sockets does not incur an unacceptable overhead. To validate this, we have designed microbenchmark-based experiments to determine the performance impact of Service Sockets. Microbenchmarks are useful to estimate the performance of the modular infrastructure as a whole. Measuring the performance of specific services/S-sockets would add few additional value to this effect.

Our benchmark establishes a connection between two peers and exchanges messages in a ping-pong fashion. We compare the performance of three configurations. (a) Both peers use standard Berkeley Sockets. (b) Both peers use an S-socket without any components ("nop"). (c) Both peers use S-sockets with a varying number of a simple data layer that copies the data as it forwards it. Note that "adding one data layer" in this setup means four additional invocations of a data layer – once up, once down the stack on each peer. We can determine the "ba-

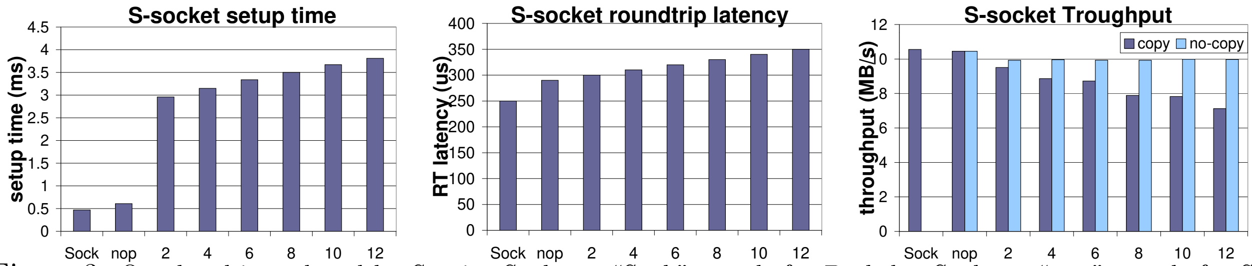

Figure 3: Overhead introduced by Service Sockets. "Sock" stands for Berkeley Sockets, "nop" stands for S-sockets with no data layers, "n" stands for S-sockets with $\frac{n}{2}$ data layers on both peers.

sic overhead" introduced by the Service Sockets *core* by comparing the performance of configurations (a) and (b). We can determine the overhead introduced due to administering multiple Service Sockets *components* by comparing the performance of configurations (b) and (c).

All experiments are conducted on two SUN Ultra 30 machines connected by a Fast Ethernet switch. We use three metrics: the *time for connection setup*, the *roundtrip latency* when sending a small message, and *sustained throughput* when sending large messages. All experiments were repeated until the measurements established a 95% confidence interval within 5% of the mean. Figure 3 presents these metrics for all three configurations.

Configuration (a) vs (b): The performance of "nop" S-sockets is very close to that of Berkeley Sockets (within 5% for throughput, and within 40 μs for latency, and within 150 μs for connection setup).

Configuration (b) vs (c): While the connection setup time for "nop" S-sockets is close to Berkeley Sockets, this is not the case for S-sockets with components. Using one component increases the connection setup time by about 2.5 ms over the "nop" case. This represents the overhead of linking a previously closed library. Increasing the number of components further illustrates the relatively small overhead of linking a previously opened library. While the connection overhead seems rather large, it is unavoidable if we want to support dynamic linking of components for improved configurability (instead of static linking). However, this overhead can be neglected in many application scenarios, since the creation of new service-level channels (i.e., S-Sockets) is a comparably infrequent operation.

The roundtrip latency increases linearly with the number of data layers. Each additional layer adds 10 μsec to the roundtrip latency. Note that this number represents four invocations of the additional data layer. Many services may not require such a symmetric configuration (i.e., the same complexity in data processing on every peer for both sending and receiving).

We use two versions of the benchmark to determine the throughput overhead. In the first version we use copy-and-forward data layers; in the second version we use only-forward data layers. The throughput decreases roughly linearly with the number of data layers in the first version while it stays constant in the second version. This showcases the performance penalty commonly associated with any application-level modular system: data movement/copying overhead between component boundaries. Service-specific requirements determine on a case-by-case basis if a component needs to perform such copy operations. For instance, if a component has the task to encrypt or decrypt data, it can not avoid extra data movement. On the other hand, a component which augments/adds/strips application-level frame headers may avoid this. It is up to the component programmer how to optimize memory allocation (desirably using the blackboard) given the requirements of the service to be performed. If our infrastructure would focus on supporting only components with very specific functionalities, e.g, conventional protocol stack components that add/strip headers [24, 11], we could provide support for optimized memory management. However, this has been done already in many application domains and would defeat our main purpose of generality.

5 Services

We have used Service Sockets to implement a wide variety of networking applications. Our experience with implementing these S-sockets has been positive. The functionality provided by the core allowed us to focus on service-specific complexity. The modular design used by an S-socket allowed us to decompose the complexity in a meaningful fashion. The well-defined core-component interfaces allow the different components to evolve separately.

Due to space restrictions, we present three select examples: (1) The `sirc` IRC S-socket that extends a legacy IRC client to provide language translation and fault tolerance. We also use `sirc` to demonstrate the control flow in Service Sockets in more detail. (2) A layered multicast S-socket [17] that handles group communication among peers with varying network connectivity and demonstrates nontrivial interactions between data processing and control handling on multiple transport channels. (3) An HTTP anycasting S-socket that selects a server from a group of replicated HTTP-based servers and demonstrates extensibility and configurability of Service Sockets. These examples do not necessarily present novel ideas on their own. We rather chose them to represent a range of different concepts and thus demonstrate the generality and flexibility of Service Sockets.

5.1 A Translating IRC S-socket

The Internet Relay Chat (IRC) allows large user groups to communicate online in chat channels. IRC clients allow users to join or leave chat channels, send messages to all or select users in the current channels, and provide filtering of unwanted messages. The underlying IRC system on the server side is distributed worldwide and manages user and channel state. If a server dies or a network partition occurs, we loose session state on the client side, not only creating security holes, but also forcing the user to reinstantiate her state.

It would be beneficial if we could use an existing IRC client and augment it to transparently provide failover, transparent to the application. We also would like to use the existing client to transparently translate between the user's language on the client side (e.g., Italian) and the chat room language (e.g., English).

To demonstrate how Service Sockets can help to achieve such a form of value-adding to legacy code, we designed the `sirc` S-socket. `sirc` intercepts and stores all state-changing commands between user and server, and reestablishes failed server connections, replaying the captured log of commands. `sirc` also uses the Babelfish tool[2] to transparently translate messages from one language into another.

We have tested `sirc` with the public domain IRC client ircII. The only modification to ircII was to replace calls to the Berkeley Sockets library with the corresponding Service Sockets calls. However, `sirc` would work with other IRC clients (even telnet) that can be linked against the Service Sockets library.

`sirc` consists of four components: two data layers, *irc_log* and *irc_savebb*, and two control components, *irc_recov* and *irc_trans*. The *irc_log* layer maintains session state by capturing user commands (e.g., `/nick`, `/join`, `/invite`, `/mode`, `/ignore`) and recording relevant state (latest assumed nickname, subscribed channels, mode, and user defined filters) on the blackboard. The *irc_savebb* and *irc_trans* components jointly provide on-the-fly translation of IRC chat messages (details follow). When a server connection is broken, *irc_recov* replays the session state after reconnecting to a working IRC server.

We exemplify the operation of Service Sockets using a sample ircII+`sirc` session, assuming the configuration file shown in Figure 4 (a):

Start of session: ircII requests a `sirc` S-socket by calling `Ssocket()` as shown in Figure 4 b). This call directly specifies the components of `sirc`. The S-socket connection setup is illustrated in Figure 5.

The `serverlang` and `userlang` parameters inform *irc_trans* about the languages used by the channel and user, respectively. In this example, outgoing (incoming) messages are to be translated from Italian to English (and vice versa). The `server` parameter specifies the translation server to be used.[3] The `reconn` parameter instructs *irc_recov* to reconnect to the same IRC server after a failure instead of choosing another server from a `sirc`-specific configuration file. These parameters hold no meaning

[2] *http://babelfish.altavista.com*

[3] In this case, the Babelfish translation service available at *http://babelfish.altavista.com*.

```
# -------------------------------------------- data layer definitions
irc_log      data://irc_log.so/
irc_savebb   data://irc_savebb.so/
# --------------------------------------- control component definitions
irc_recov    control://irc_recov.so/server='my.irc.com'
irc_trans    control://irc_trans.so/
```

a) Configuration file

```
Ssocket("irc_log|irc_savebb|irc_recov(reconn='same server')|\
   irc_trans(srvlang='english',usrlang='italian',server='babelfish.altavista.com')",
   AF_UNIX, SOCK_STREAM, 0);
```

b) S-Socket creation

Figure 4: Sample configuration file and S-socket creation for `sirc` components.

to the core and are only interpreted by the components.

The configuration of the `sirc` S-socket proceeds as detailed in Section 3. The initialization routines of all four components are called and result in allocating an IRC command log (*irc_log*); allocating a message buffer on the blackboard (*irc_savebb*); registering a trigger for assignments to this message buffer and associating the `translation()` event handler with this trigger (*irc_trans*); and registering the `fault_recovery()` handler for TCP connection failures (*irc_recov*). We do not need a special service setup event handler for this case, i.e., the core just maps `Sconnect()` to `connect()` by default. The state of the `sirc` S-socket after initialization is illustrated in Figure 5.

User → Server Communication: ircII calls `Ssend()`, which passes the data down the data layer stack. First, `send()` in *irc_log* checks if this is a relevant IRC command and if so, updates the state stored on the blackboard. Next, `send()` in *irc_savebb* saves the data to the message buffer. This triggers the `translation()` event handler which communicates with Babelfish through a WebL script [13]. When control returns to `translation()`, it saves the translated message on the blackboard and control returns to *irc_savebb*, which allows it to proceed to the network.

Server → User Communication: ircII calls `Srecv()`, which receives the data from the network, and then passes it up the data layer stack. `recv()` in *irc_trans* performs the (reverse) translation just like `send()`. Since only outgoing chat messages are inspected, *irc_log* does not need to implement `recv()`.

Server Failure: Whenever the core executes an `Ssend()` or `Srecv()` and detects the network event of a broken TCP connection, it passes control to *irc_recov* which tries to reconnect to the same IRC server. After a new connection is established, the server state captured by *irc_log* is replayed to the server.

Note that *irc_savebb* and *irc_trans* are independent of the other two components; it is possible to remove them from `sirc` resulting in a logging-only S-socket or to use them independently to build a translation-only S-socket.

5.2 A Layered Multicast S-socket

We have chosen this example to illustrate the architecture of an S-socket that employs a variable degree of transport-level channels. Layered multicast is a technique to deliver multimedia content to a heterogenous group of receivers [17]. 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 missing [17]. The lowest layer is called the *base layer* and the higher layers are called *enhancement layers*. The service provider sends each layer on a separate multicast group. Each receiver can individually tailor the service to its network connectivity by selecting a subsequence of layers to receive. 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 setup, the server partitions the data stream

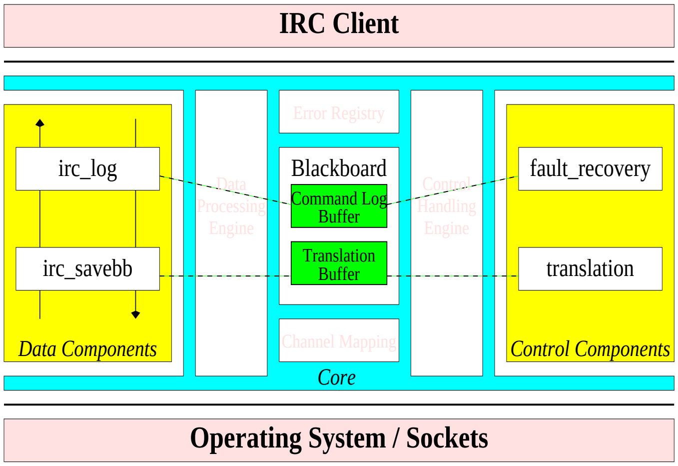

Figure 5: `sirc` S-socket after initialization.

on each multicast group into a sequence of data frames. Each frame includes a sequence number that clients use to combine frames from different layers. Our layered multicast S-socket consists of two control components for service setup and layer adjustment, and one data layer.

Service setup: This component contacts the server using a TCP connection and requests information for the desired session (number of groups used, address and expected bandwidth requirement for each group). Since we have no information about the available network bandwidth, we initially join all multicast groups.

Data processing: The core receives from all selected groups and presents a sequence of buffers to the data layer. The data layer maintains a sliding window based on the sequence numbers of frames. Each slot in the window is reserved for frames with the same sequence number. 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 sequence number have been received. In this case, the data frames with this sequence number are combined and the result is passed up to the application. If the window fills up, the data layer discards the oldest incomplete slot. 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 for more than t seconds or lower than a low-water mark for more than $2t$ seconds, it writes to a trigger on the blackboard which calls the layer adjustment handler in the control layer.

Layer adjustment: This component manages all changes in the set of layers subscribed to. The current data-loss rate and configuration info is retrieved from the blackboard. Depending on the high- and low-water marks and the data-loss rate, the number of multicast groups is incremented or decremented and the S-socket state is updated with the new set of received groups, the updated configuration info and the reset sliding window frame buffer.

Note that the trigger mechanism allows us to easily change the layer adjustment scheme from the layered multicast approach (select subset on n layers) to a simulcast (select one out of n layers) without modifying the data layer.

5.3 An HTTP Anycasting S-socket

We have selected this example to illustrate the configurability of applications developed using Service Sockets. Anycasting allows a client to select a server for a replicated service [2, 20]. 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: (a) Using `gethostbyname()` to 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. (b) Using a scheme similar to the "/robots.txt"-based scheme used by HTTP servers to control the operation of web crawlers. 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. (c) Using a general-purpose mirror service that returns a set of servers that provide the requested URL.

Schemes (b) and (c) can optionally provide a time-to-live for mirroring information that can be cached on disk by the anycasting code as appropriate.

Selecting a server: (a) Using the top-level domain of the server domain names to filter out servers known to be far away. For this purpose, it considers `.com`, `.edu`, `.mil`, `.net` and `.org` as equivalent. (b) Using TCP connection time as the performance metric. (c) Using TCP connection time plus time to issue the HTTP request "`GET /`" and receive the reply as the performance metric.

We have developed an HTTP anycasting S-socket based on these different techniques. The requested URL and the choice in technique of finding the set of servers and selecting a server is specified as parameters in the *serviceSpec*. The anycasting functionality is completely implemented in the service setup event handler. This allows for easy integration with other service-level functionality to form value-added S-sockets utilizing anycast.

6 Related work

The idea of a common and easy-to-use interface to a diverse collection of resources has had a long and successful history in operating systems, as it allows programmers to operate on a wide variety of resources without having to learn and use a large variety of potentially incompatible interfaces. Berkeley Sockets was designed to interact with a variety of network protocols (twenty-five alone are listed in `<sys/socket.h>`), and has often been used as the interface of choice for new protocols and devices. Another prominent example is the use of the file system interface to manipulate and interact with disk-resident files, devices, protocols (`/dev/{tcp,udp}`) or processes (`/proc`). In the following, we relate Service Sockets to approaches advocating modular techniques for network programming, to specialized network libraries built on top of sockets, and to distributed service frameworks.

Modular Networking: Similar to the data layer stack in Service Sockets, STREAMS [24] is used to interpose a stack of processing modules between a device and an application. Both approaches allow processing modules to be chained into a bidirectional data path and (de-)multiplexing from (to) the data path. Unlike Service Sockets, STREAMS modules reside within the kernel which limit configurability for unprivileged users. More importantly, unlike Service Sockets , STREAMS provides only limited scope for control handling[4]. As the examples presented in the introduction indicate, most of the increase in client complexity is due to control handling. Other module-based frameworks for implementing network protocol stacks on end hosts include the x-kernel [11] and Scout [19]. For each message sent or received, Scout determines the path among the components it has to pass. Once this path is determined, the processing code and scheduling can be optimized. The major thrust in Scout is to improve performance, not to provide a general high-level networking interface. Also, unlike Service Sockets, Scout modules reside in kernel space.

Application-class Specific Networking Libraries: The SmartSockets library from Talarian [28] enables publish-subscribe applications. RTP libraries [15] enable applications requiring real-time data delivery. The event-driven ACE communication library [25] uses object-oriented "wrapper facades" to extend the socket interface. Privacy Enhanced Sockets [30] provides network privacy for stream sockets. Such libraries are focused towards their application class and provide idiosyncratic and usually incompatible interfaces. On the other hand, Service Sockets leaves the Berkeley Sockets interface unchanged except one string argument added to `Ssocket()`.

General Distributed Object Frameworks: Frameworks like CORBA [21] or Jini [27] take a different approach to abstracting the details of interacting with services. Both automate service location and service initialization by using fixed service-independent techniques. However, they do not provide a uniform interface for interacting with diverse services.

7 Discussion

The current prototype implementation of Service Sockets does have its limitations: some that are in-

[4]STREAMS only defines a limited number of control messages that are sent up and down the stack just like regular data.

herent to any modular, user-level framework (like isolation and performance issues), and some that are inconveniences that can be fixed easily.

The problem of isolation is rather perceptual and grounded in the fact that the close modeling of Service Sockets after the BSD socket interface lulls users that S-sockets behave identically to standard BSD sockets. In reality, Service Sockets is a user-level library system and may create unexpected pitfalls. For instance, a Service Sockets component may – just like any library we run without a correctness proof – cause an exception, or simply enter an infinite loop. While techniques like watchdogs or software fault isolation [29] may partially alleviate such problems, it seems pointless to hope for a guaranteed 100% BSD compatible extensible interface. We rely on the cooperation of service designers and only enforce certain consistency assertions of S-socket state after components return control back to the core.

Dynamic linking provides increased configurability but can significantly increase the creation time of an S-socket. However, we focus in this paper on networking clients which generally open and close service-level connections relatively infrequently. While we have not used Service Sockets yet to implement networking servers or proxies, it is clearly possible to do so. Since dynamic linking would become a real performance issue on many server-side applications we can extend Service Sockets to allow static linking of components for cases when performance matters more than configurability.

Copying due to component boundaries can significantly affect performance in the presence of the following conditions: (1) very fast networking, (2) relatively weak computing power, and/or (3) service-specific tasks performed by the components requiring very few resources[5]. However, if an application scenario meets these conditions and is sensitive to performance, component programmers have the option to optimize memory management tailored to their application domain. For instance, by implementing equivalents to `mbuf` structures (traditionally used for in-kernel protocol stacks) on the blackboard. If instead, such optimized memory management would be part of the core functionality, certain application domains would benefit, but others would not (e.g., `mbufs` don't make sense for compression or encryption tasks). Therefore we choose a simple common denominator concerning the memory management functionality that is provided by the core: the blackboard – all application domains can use it as-is or build abstractions on top of it. Such abstractions would not be part of the core but can be reused to build various components in a given application domain. This approach requires additional effort from component developers for performance-critical applications, but retains the generality of Service Sockets.

[5]Note that the microbenchmark in Section 4 was setup to meet these conditions, demonstrating a sort of worst-case "modularity overhead".

As an S-socket is configured only once at creation time, dynamic changes in the set of components are currently not supported. Consider a client connecting to a variety of servers that might use any combination of stream encryption, compression, or special encodings. Currently, Service Sockets requires to put every "potentially-used" component in the data layer stack at configuration time. It would be useful to dynamically add only the "really-used" components to the stack that are agreed upon by all peers in a negotiation phase during service setup. This could be easily accomplished by allowing the control components to push/pop data layers to/from the stack.

While the idea behind Service Sockets is to simplify applications by encapsulating communication with a service into the Service Sockets framework, applications might sometimes still want to exert some kind of control on an S-socket. One simple way to achieve this is to use out-of-band data. As an application might already use out-of-band data for other uses, a better way may be to implement a service-level extension for the socket-option interface.

8 Concluding Remarks

In this paper, we have proposed the *Service Socket* abstraction. Service Sockets allows service designers to decompose new network functionality into data and control components that can be installed in an extensible framework that is implemented completely at the user level. This allows existing applications to use new functionality encapsulated in third-party extensions with minimal or no changes.

In essence, a Service Socket, while retaining the conceptual simplicity, familiarity and semantics of the BSD socket interface, is really a communication channel to a *service* instead of a particular *host*. Using such an extensible and configurable service-level framework allows a large set of application domains to leverage one common interface.

To demonstrate the feasibility and utility of our approach, we have implemented a number of useful and representative S-Sockets. To show how to use Service Sockets to extend a legacy application, we converted an existing IRC client into a turbo-charged client that memorizes and replays session state after server outages and translates messages between the user's and the chat room's language. To demonstrate nontrivial interactions between data processing and control handling on multiple transport channels, we have implemented a layered multicast S-socket. To demonstrate extensibility and configurability of Service Sockets, we have constructed the HTTP anycasting control component which can add extra service setup functionality to any existing HTTP-based service.

As our experimental results suggest, the overhead of Service Sockets is acceptable for many application classes. The framework does not prevent component designers to optimize the overhead tailored to specific performance-sensitive application classes.

We are aware of several interesting ways to extend Service Sockets and conduct further research. The current prototype can be augmented with extra flexibility, for instance, in-core support for dynamic component reconfigurations. We can also build out-of-core abstractions to allow improved performance for certain application domains, for instance, `mbufs` built on top of the blackboard for (application-level) protocol stacks.

Another research direction is to simulate BSD sockets even further – by intercepting system calls to `socket()`, `recv()`, `send()`, etc through the `/proc` interface [6]. Thus, we could run applications completely unmodified, similar to the Ufo global file system [1].

Finally, it would be interesting to develop a declarative specification for control handling in Service Sockets in the spirit of the Click modular router [18]. Such a specification could enable the Service Sockets framework to reason about the correctness of the control flow, obviating the need for complex explicit assertions about the internal state of an S-socket, especially when the number of components increases.

References

[1] A. D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Scheiman. Extending the operating system at the user-level: the ufo global file system. In *Proc. of the USENIX Annual Technical Conference*, pages 77–90, Jan. 1997.

[2] 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.

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

[4] D. Clark and D. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proc. of SIGCOMM*, 1990.

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

[6] R. Faulkner and R. Gomes. The process file system and process model in UNIX System V. In *Proc. of the Winter 1991 USENIX Conference: Dallas, TX, USA*, Jan. 1991.

[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*, 1998.

[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*, 1995.

[9] A. Fox, S. Gribble, E. Brewer, and E. Amir. Adapting to network and client variability via on-demand dynamic distillation. In *Proccedings of ASPLOS VII*, 1996.

[10] J. Guyton and M. Schwartz. Locating nearby copies of replicated internet servers. In *Proc. of SIGCOMM*, 1995.

[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*, 1994.

[13] T. Kistler and H. Marais. WebL — A programming language for the Web. In *Proc. of the 7th International World Wide Web Conference*, 1998.

[14] I. Kouvelas, V. Hardman, and J. Crowcroft. Network adaptive continuous-media applications through self-organised transcoding. In *Proc. of NOSSDAV*, 1998.

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

[16] J. Mahdavi and S. Floyd. TCP-friendly unicast rate-based flow control. Technical note sent to the end2end-interest mailing list[6], Jan 1997.

[17] S. McCanne, V. Jacobson, and M. Vetterli. Receiver-driven layered multicast. In *Proc. of SIGCOMM*, 1996.

[18] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. In *Proceedings of the 17th ACM SOSP*, 1999.

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

[20] A. Myers, P. Dinda, and H. Zhang. Performance characteristics of mirror servers on the internet. In *Proc. of Infocom*, 1999.

[21] Object Management Group, Inc. *CORBA 2.3.1/IIOP Specification*[7], 1999.

[22] 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.

[23] 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*, 1999.

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

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

[7] Available at *http://www.omg.org/library/c2indx.html*

[25] D. Schmidt. Ace: an object-oriented framework for developing distributed applications. In *Proc. of the Sixth USENIX C++ Technical Conference*, 1994.

[26] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RFC 1889: RTP: A transport protocol for real-time applications, Jan. 1996.

[27] Sun Microsystems, Inc. *Jini Technology Architectural Overview*[8], 1999.

[28] Talarian: Our Products: SmartSockets[9], 1999.

[29] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Proc. of SOSP 1993*, pages 203–16, 1993.

[30] A. Zuquete and P. Guedes. Transparent authentication and confidentiality for stream sockets — ensuring private network communications for Unix and Windows systems. *IEEE Micro*, 16(3):34–41, June 1996.

[8] Available at *http://www.sun.com/jini/whitepapers/-architecture.pdf*

[9] Available at *http://www.talarian.com/products/-smartsockets/smartsockets.shtml*