Wrens: A Framework for Rapidly Evolvable Network Services

Anurag Acharya Maximilian Ibel Michael Schmitt
Mathias Koelsch Davor Magdic Ben Smith Murat Tuncer

Department of Computer Science
University of California, Santa Barbara

TRCS99-31
September 1999

Abstract

In this paper, we describe the *Wrens* framework for rapidly evolvable network services. The goal of Wrens is to provide the abstractions and the infrastructure that facilitate rapid development, extension and deployment of network services. The design of Wrens is based on four principles. First, the communication structure of services should be specified explicitly in a form that can be understood by the infrastructure. This allows the infrastructure to automatically instantiate complex services, detect many kinds of failures and (in many cases) automatically repair them. In addition, it allows service designers to rapidly prototype services by plugging together existing service components. Second, complex services should be constructed as a collection of interacting service modules that communicate via named channels. Service modules written in this manner are easier to plug together since the interactions between such modules are limited to explicitly named channels. Third, the level of interaction between end-user applications and the communication infrastructure should be raised – instead of being limited to asking for a communication channel to a specific server, applications should be able to request a named service. This facilitates service evolution as the structure and the implementation of a service can be changed without changing the requesting application. Furthermore, this provides an opportunity for the infrastructure to transparently handle many kinds of faults. Finally, the specification and implementation of services should use symbolic names wherever possible. Symbolic names provide a level of indirection that allows the infrastructure to transparently adapt a service to varying levels of resource availability with relative ease.

We describe our prototype implementation of the Wrens framework and report our experiences with developing and deploying a diverse suite of services. Our experience has been positive. We were able to develop these services in a short time. We were also able to develop useful extensions for several of these services in the matter of a few hours. We also present an evaluation of the overheads introduced by the Wrens framework and the costs of end-to-end operations.

1 Introduction

The Internet is currently in the process of transforming from an infrastructure which primarily transports data to an infrastructure which provides a wide variety of services to an increasingly diverse clientele. A large number of traditional services are already online (e.g., booksellers, banks, libraries, brokerages, and travel services). In addition, new classes of services are made possible by the rapid growth in the connectivity and the volume of information available (such as search engines, customized news tickers, price trackers, chat rooms etc).

Deep as this transformation is, we believe that a potentially deeper transformation is on the horizon. Today, users access services from desktop machines. Current trends, however, indicate that in a not-too-distant future, networked computing appliances will become a pervasive part of our environment [5, 18, 19, 25].[1] Environments with embedded devices are likely to be permeated with network services – they would need to be if they are to achieve their full potential. The ability to access services *in-situ* is likely to catalyze entirely new classes of services. For example, an information aggregation service that combines continuously updated information about parking, traffic and your geographical position as you drive to work and plans a continuously updated route; or an attribute-based messaging service that can send a message to everyone who is driving a large vehicle north on I-95 and is within thirty miles of the George Washington Bridge. The variety of access devices and services in such scenarios indicate a need for service infrastructures that facilitate rapid evolution and prototyping.

In this paper, we describe the *Wrens* framework for rapidly evolvable network services. The goal of Wrens is to provide the abstractions and the infrastructure that facilitate rapid development, extension and deployment of network services. The design of Wrens is based on four principles. First, the communication structure of services should be declaratively specified in a form that can be understood by the infrastructure. This allows the infrastructure to automatically instantiate complex services, detect many kinds of failures and (in many cases) automatically repair them. In addition, it allows service designers to rapidly prototype services by plugging together existing service components. Second, complex services should be constructed as a collection of interacting service modules that communicate via named channels. Service modules written in this manner are easier to plug together since the interactions between such modules are limited to explicitly named channels. Third, the level of interaction between end-user applications and the communication infrastructure should be raised – instead of being limited to asking for a communication channel to a specific server, applications should be able to request a named service. This facilitates service evolution as the structure and the implementation of a service can be changed without changing the requesting application. Furthermore, this provides an opportunity for the infrastructure to transparently handle many kinds of faults. Finally, the specification and implementation of services should use symbolic names wherever possible. Symbolic names provide a level of indirection that allows the infrastructure to transparently adapt a service to varying levels of resource availability with relative ease.

Our prototype implementation of Wrens runs on Solaris, Linux and the PalmOS.[2] We have used the Wrens prototype to implement a diverse suite of services including `porthole`, a location-dependent information service which combines parking, traffic and facility location information for mobile clients; `customcast` (customized multicast), a filtering news ticker which adaptively distributes customized news content over a set of multicast groups; `nuz`, which transcodes and caches news articles distributed by `customcast` on behalf of PalmPilot users; `SMP3`, which manages on-demand delivery of streaming MP3 audio; `selectcast` (selective multicast), which provides a lightweight one-time multicast of messages to a sender-selected receiver-group; and `plamchat`, which allows PalmPilot users to securely chat with peers on other PalmPilots or on workstations.

[1] Cell phones from Nokia [16], Motorola [13] and NTT [2] already provide (or will soon provide) Internet access and the ability to run a variety of applications. Car-based PCs with the form-factor of a car radio and wireless connectivity are already available from Clarion [4] and will soon be available from other vendors [11, 27]. A variety of wireless-connected devices that can be embedded into everyday environments are also expected to become available in the near future [3].

[2] We are currently working on a WindowsCE port.

Our experience building these services has been extremely positive. We were able to develop, test and deploy these services quite rapidly. The first pair of services, `porthole` and `customcast`, took about three weeks each to develop; the other four took much less time – between a couple of hours and four days. In addition, we were able to develop useful extensions for several of these services in the matter of a few hours.

We first give an overview of the Wrens framework and its design principles. Next, we describe its main components and how they fit together. Then, we present short descriptions of the services we have implemented using Wrens. We then evaluate the performance of Wrens and its ability to transparently repair a service using a suite of micro-benchmarks. Finally, we discuss related work and present conclusions.

2 Overview

The design of Wrens is centered around four tightly interwoven concepts.

Declaratively specified communication structure: Wrens services are specified using the *Wrench* specification language (see Section 3.3 for details). It allows service designers to specify the desired features of service components and communication channels as annotations in service description graphs. A Wrench service specification consists of a graph – the nodes being the service components and the links being the communication channels between the service components. Specifying the structure of a service separately instead of hardcoding it in individual modules makes it easier for service designers to modify and extend existing services – to add/remove/replace a service component, or to change the protocols being used.[3] Having access to declarative service specifications also allows Wrens to automatically instantiate complex services, detect many kinds of failures, and in some cases, repair a failed service by restarting a component (possibly at a different location).

Wrench supports dynamic evolution of services by allowing a newly created service to be merged into a currently operational service. This allows new value-added services to be built on top of existing existing service. To the original service, the new service being merged appears just like any other client, whereas to the new service, already operational service appears like a part of itself. This easy evolution is possible due to the stream-based nature of the service components. Since service components are not able to determine the source or the sink of a stream, two services can be transparently plugged together.

To provide seamless integration with clients and servers that are unaware of Wrens, Wrench allows for *legacy links* at the edge of service description graphs. Legacy links can be: *accept* links which allow unmodified client applications to connect to (and possible initiate) a Wrens service; or *connect* links which allow Wrens services to connect to unmodified servers. Legacy links allow Wrens services to be build on and to co-exist with existing services.

Stream-based service components: Wrens provides a stream-based programming model for service components. All communication channels of a service component are presented to it as pre-created streams. Streams are created by the Wrens framework as a part of instantiating the service. Individual components can neither create new streams nor can they determine the source or the sink of a stream. Service modules written in this manner are easier to plug together since the interactions between such modules are limited to explicitly named channels. Making the com-

[3]We were able to make useful extensions to some of our services just by changing the service specification, with no changes to the modules at all.

munication channels explicit in this manner also makes it easier to restart service components if and when they crash.

Active directories: Wrens uses symbolic names for services, service components, communication channels and service locations. As a result, lookup is the fundamental control construct in Wrens. To support dynamic evolution of services, Wrens provides *active directories*. Active directories allow a function to be executed as part of the lookup operation. This allows the lookup operation to return different results based on the identity/location of the client, parameters specified as a part of the lookup operation, the previous sequence of lookup requests etc. The dynamic nature of the lookup for an *active directory* allows Wrens to automatically adapt (or compose) a service based on information about server load, client's location and connection to the service infrastructure, client's access device, intended set of peers etc.

Service sockets: Wrens provides *Service Sockets* to raise the level of interaction between end-user applications and the communication infrastructure. *Service Sockets* are a generalization of the BSD-style sockets. They differ from BSD-style sockets in that they allow applications to request a *service* instead of a communication channel to an end-point. They resemble BSD-style sockets in that they can transparently detect and handle many classes of faults. They also provide a programming interface that closely resembles the BSD socket interface.

Service Sockets functionality common to all services (e.g., selecting a server when initiating service, detecting when a peer is no longer available, re-establishing a failed service etc) is provided by a user-level library that can be linked into client applications. In addition, Service Sockets allows service-specific functionality to be seamlessly integrated as application-level layers on top of the IP stack. To allow different layers to be stacked with relative ease, Service Sockets requires each layer to implement a common interface (which we refer to as the *Vsocket* interface). Each layer is implemented by a dynamically linked library.

3 The Wrens framework

The Wrens framework consists of five parts: (1) the Wrens execution environment for service components, (2) the *Wrench* service description language, (3) the stream-based programming model for service components, (4) the Vsocket interface for stacking processing layers on top of the IP stack, and (5) the *ActiveLDAP* directory for resolving symbolic names. In this section, we describe these parts. In the next section, we describe how these parts fit together to implement distributed instantiation of a service with multiple service components and distributed fault-recovery when a service component goes down.

3.1 Execution environment for service components

We believe that in a not-too-distant future, there will be two classes of network services with different scope and usage patterns. The first class of services would be global information and commerce services used by a very large client-base (e.g., brokerages, auctions, search engines etc). To power this class of services, we believe that clusters of workstations would be a suitable platform – as they can be incrementally scaled and upgraded, are easy to replicate for high availability and provide an attractive price-performance ratio. We share this belief with several other researchers [1, 6, 7],

The second class of services would be localized in nature and would provide dynamically changing information of interest to a small client-base. Examples of such services would include parking lot availability, local transportation schedules, construction zones, etc. The growth of such services is likely to be fueled by the increasing penetration of our environment by embedded networked devices. For many of these services, much simpler platforms would suffice. In fact, many of these services may exist only seasonally (mall parking for Christmas shopping or local festival) or on demand.

The Wrens execution environment has been designed with both classes of services in mind. It provides a loose and flexible organization without centralized component so that a platform for Wrens services to be set up quickly. Since faults can be expected to be relatively common in a loose organization such as this, Wrens runs a distributed fault detection and recovery algorithm which monitors each service and tries to transparently repair faults (see Section 4.2).

To become a Wrens-capable node, a workstation runs two administrative processes, the `dispatcher` and the `watchdog`, and makes itself and its capabilities (i.e., the service components it can run) known to the appropriate lookup servers. The `dispatcher` is the main administrative process on a Wrens node: it receives requests for service instantiation directed to this node, resolves symbolic names in the service description (if any) and instantiates local service component by starting a service process (or notifying a service process if one already exists). It monitors the state of the service processes and restarts any process that crashes. The `watchdog` process monitors the status of the `dispatcher` using a heartbeat mechanism. If it does not receive two consecutive heartbeats, it kills the dispatcher process and creates a new one.

A service process is created by the dispatcher when the first instance of a service component is requested. When a service process starts up, it creates a *master* thread. The *master* thread is a part of the Wrens framework and is implemented by the `librens.a` library linked in with all service components. The master thread performs all administrative operations for a service process. It receives requests for instantiating service components from the dispatcher. It creates the environment for every service component by creating all its streams and by spawning a dedicated thread. It schedules service components when data is available on their streams. The master thread also monitors the health of individual components, restarting them as need be. Finally, it participates in distributed instantiation and fault-recovery protocols for end-to-end service operation (see Section 4).

The `dispatcher` maintains a connection (currently implemented using a Unix pipe) to the master threads of all service processes. This allows the dispatcher to rapidly forward instantiation requests as well as to keep track of the health of the service processes. If a connection is broken, it assumes a fault has occurred in the associated service process; therefore it kills and restarts it.

3.2 Stream-based programming model for service components

Wrens provides a stream-based programming model for service components. All communication channels of a service component are created on its behalf by Wrens. Service components can neither create new streams nor can they determine the source or the sink of a stream. Furthermore, they should not create other communication channels (sockets/pipes/etc). Accepting these restrictions provides three advantages for service components. First, this allows Wrens to automatically instantiate complex services. Second, service components written in this manner are easier to plug together since the interactions between such components are limited to explicitly named channels. Third, having Wrens manage all communication channels allows it to detect faulty service components/channels, and, in some cases, to transparently recreate them. Figure 1 presents the programming interface for service components to interact with streams. In addition to the customary read and write operations, the interface includes two procedures, `streamClose` and

Procedure	Parameters	Description
`streamRead`	`stream`, `buffer`, `buflen`	read at most `buflen` bytes from `stream` into `buffer`.
`streamWrite`	`stream`, `buffer`, `buflen`	write `buflen` bytes from `buffer` to `stream`.
`streamClose`	`stream`	used to indicate voluntary termination of this service component to the peer at the other end of the stream.
`streamFault`	`stream`	used by a service component to inform Wrens about a stream that may have been involuntarily terminated.

Figure 1: Programming interface for Wrens service components

`streamFault`, to facilitate fault detection. `streamClose` is used to notify a peer about voluntary fault-free termination of a service component. This avoids triggering the fault-recovery mechanism during service shutdown. `streamFault` is used to inform Wrens about a stream that may have been involuntarily terminated. This is needed for streams that use non-connection-oriented transport protocols (UDP/multicast). For such streams, Wrens has no service-independent way of detecting if the peer at the other end has crashed. `streamFault` allows services that use such streams to implement a service-specific way of detecting faults (timeouts, periodic heartbeat messages or a service-specific protocol) and still use the service-independent fault-recovery mechanisms provided by Wrens (see Section 4.2.2).

Service components execute in an event-driven manner. Wrens generates four kinds of events: `init`, `rebind`, `abort` and `process`. Accordingly, service components are written as a set of four event handlers - one per event. An `init` event is generated when a new instance of a service component is created. It has two parameters: the set of streams created by Wrens on behalf of the service component and the service parameters specified by the requesting client (e.g., the client's interest profile for a customizing news ticker). Wrens guarantees that `init` is the first event to be delivered to a newly created service component. A `rebind` event is generated every time there is a change in the environment of a service component (e.g., if a peer dies or new connections are merged into an existing one). It is parameterized by the set of streams whose state has changed (either created by Wrens or destroyed by the termination of a peer). In addition, if the `rebind` occurs due to new connections being merged into an existing service component, the service parameters for the new connections are provided as an additional, optional parameter. It has one parameter – the stream which has the new data. An `abort` event is generated whenever the service component is to be terminated (e.g., when the entire service is being shut down). A `process` event, which is the common case, is generated whenever there is data available for reading on any of the input streams of the service component.

3.3 The Wrench service specification language

A Wrench service specification consists of a header and a sequence of *service component blocks*. The header contains information pertaining to the entire service as well as book-keeping information such as Wrench version and the number of following component blocks. Each service component block specifies the desired features of a single service component, including the streams it needs to communicate with its peers or external entities. Figure 2 (a) presents the skeleton of a service component block. Figure 2 (b) presents a sample Wrench specification for a transcoding service that connects to an existing streaming MP3 service using a TCP connection, down-samples the

```
component componentID location
    stream 0 protocol tags peer1
    ...
    stream N protocol tags peerN
    instance = instanceID
    maxmerges = number
end
```

(a) component block skeleton

```
header
  version 1.1
  components 2
end
component 0 ldap://transcode.com:6767/downsample%intype=MP3
  stream 0 tcp in out 1 1 /* component 1, stream 1 */
  stream 1 mc in out local://224.0.0.17:4444/
  instance = -1 /* this is a new component */
end
component 1 local://MP3play.com:9897/SMP3mgr
  stream 0 tcp in out
          ldap://MP3dir.com:3435/GetMusic%type=classical
  stream 1 tcp in out 0 0 /* component 0, stream 0 */
  instance = -1 /* this is a new component */
  maxmerges = 10
end
```

(b) A sample Wrench specification

Figure 2: Wrench illustrations.

data and multicasts it to a local group. The corresponding service graph is depicted in Figure 3.

The *location* specification for a service component can either be a *(machine-name, port-number)* pair or a symbolic name which is transparently resolved by Wrens. A symbolic name in Wrens consists of three parts: the directory where it can be looked up, an anchor symbol which is used as the key for the lookup and a set of parameters which are passed to the directory as a part of the lookup operation. This structure has been designed to allow service designers to customize lookup directories with relative ease. Including the directory as a part of the symbolic name allows easy redirection of lookup operations; including parameters as a part of a lookup operation allows service designers to take advantage of *active directories* - the parameters specified in this manner are passed to the function that is run as a part of the lookup operation at the directory. In Figure 2 (b), the location for the first component is specified as a symbolic name to be looked up at the LDAP server `transcode.com:6767`. This lookup operation is intended to answer the question – *at which*

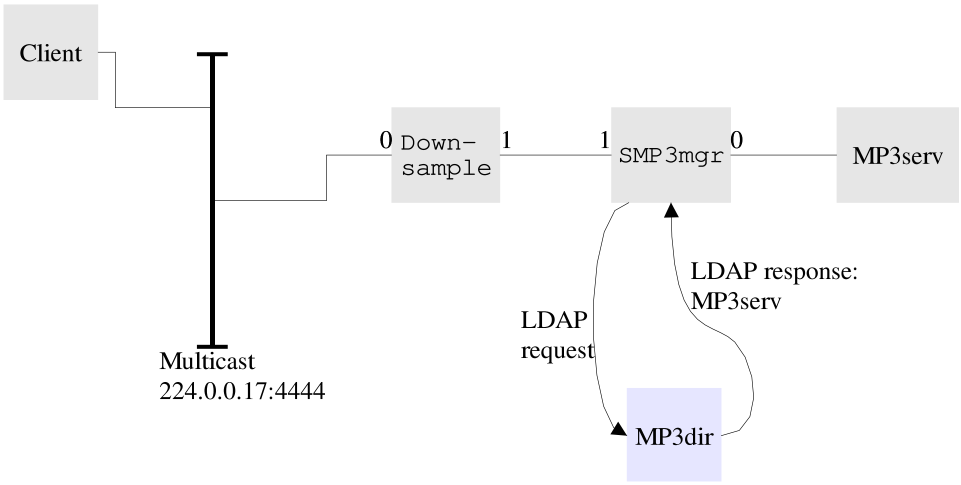


Figure 3: Structure of the service specified by the Wrench specification in Figure2.

location can I instantiate a service component that can downsample MP3 data?.

The *protocol* field for a stream specifies the transport protocol to be used. Currently, this field can be `tcp`, `udp`, or `mc` (multicast). The *tags* for a stream specify various attributes of the stream. Currently, two attributes are supported: direction and instantiation-mode. The direction attribute can be `in`, `out`, or `in out`: data can flow *in* to a component on a stream marked `in`; data can flow *out* from a component on a stream marked `out`. The instantiation-mode attribute can be `lazy` or `eager`. If a stream is marked `lazy`, only a handle is created for the stream during service instantiation; the actual communication channel (socket/pipe) is not created until first use. Otherwise, the stream is created during service instantiation (which is the default).

The *peer* field for a stream specifies what the other end of a stream is to be connected to. There are three alternatives: (1) another service component block in the same Wrench specification, (2) a *(machine-name, port-number)* pair, and (3) a symbolic name. The first alternative is used to specify the connections between different service components making up a single service. The second and third alternatives are referred to as *legacy links* and are used to provide seamless integration with clients and servers that are unaware of Wrens. Legacy links allow Wrens services to build on and to co-exist with existing services. The specification in Figure 2 (b) uses two legacy links – the multicast group `224.0.0.17:4444` and a symbolic name that needs to be looked up at `MP3dir.com:3435`. The lookup for the symbolic name takes two parameters – the name of the MP3 server (`GetMusic`) and the type of music (`classical`).

Wrench supports dynamic evolution of services by allowing new services to be merged into currently operational services. The service component blocks in a Wrench service specification can refer either to service components that need to be instantiated as a part of setting up this service or to service components that already exist. The *instance* field is used to differentiate between the two classes of service components. For service components that are to be instantiated anew, the value of the *instance* field is specified to be `-1`. For existing service components that are to be included in the new service, a non-negative value is specified which identifies a particular instance of the service component. Figure 4 provides an example. In this example, a user who does not have multicast support wants to obtain a 12Kbps MP3 stream using an existing instance of a streaming MP3 manager (the `SMP3mgr` instance from Figure 3), The first component block in this specification specifies a new service component (*instance = 1*) which downsamples the MP3 stream provided by

```
header
  version 1.1
  components 2
end
component 0 ldap://transcode.com:6767/downsample%intype=MP3%rate=12K
  stream 0 tcp in out 1 1 /* component 1, stream 1 */
  stream 1 self /*stream to client*/
  instance = -1 /* this is a new component */
end
component 1 local://MP3play.com:9897/SMP3mgr
  stream 0 tcp in out 0 0 /* component 0, stream 0 */
  instance = 42 /* this is an existing component */
end
```

Figure 4: Example of a Wrench service specification with new and existing service components.

an existing streaming MP3 manager (*instance = 42*). Structure of the resulting merged services is shown in Figure 3.3.

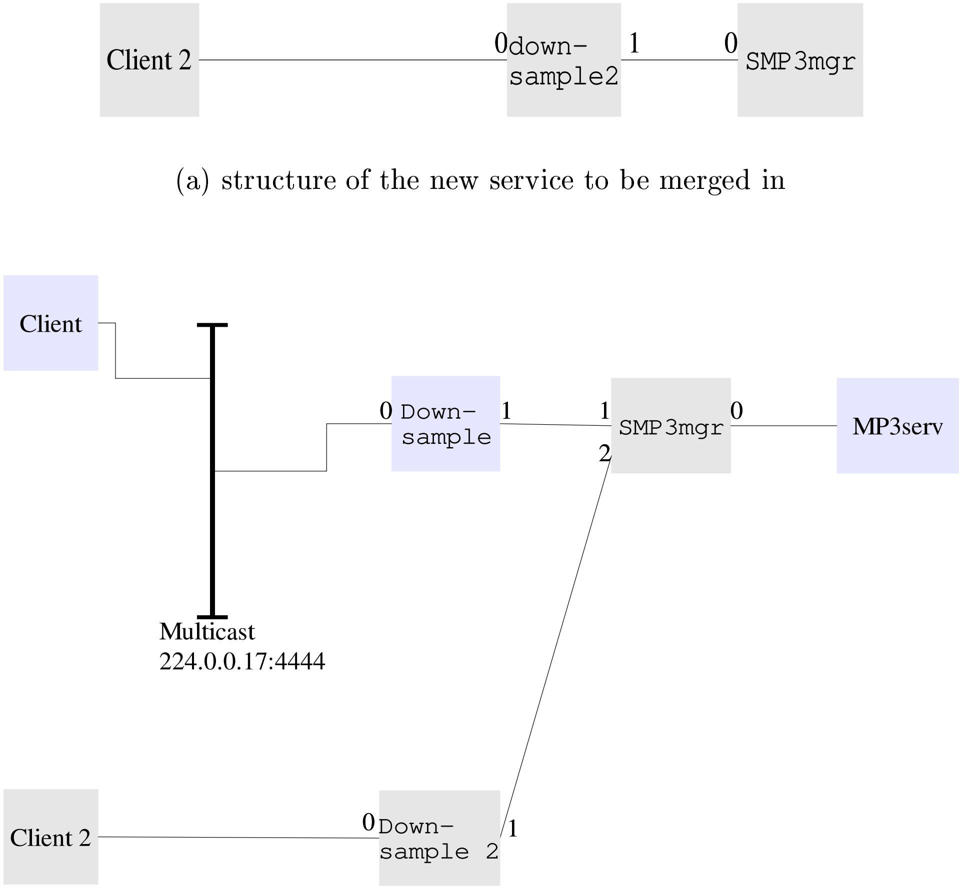
To instantiate services that contains both new and existing service components, Wrens needs to extend the set of communication channels for existing service components. Furthermore, it needs to notify service components whose environment is being modified in this manner, To achieve this, Wrens generates a `rebind` event and dispatches the corresponding event handler in the service component (see Section 3.2). The set of streams created for this operation is passed as an argument to the handler, This allows the service component to update its internal state as well as become aware of its new peer(s), Note that since rebinds can occur at any time, service components that can be extended in this manner to ensure that access to critical data elements are thread-safe.

To limit the services that can utilize a single service component, Wrens allows a service component block to include a *maxmerges* which specifies an upper bound on the number of services that can utilize a single service component. In the above example, at most 10 services can utilize an `SMP3mgr` service component (specified as *maxmerges = 10* in Figure 2).

3.4 Service Sockets

Wrens provides *Service Sockets* to raise the level of interaction between end-user applications and the communication infrastructure. Service Sockets allow applications to request a *service* instead of a communication channel to an end-point. This facilitates service evolution as the structure and the implementation of a service can be changed without changing the requesting application. Furthermore, this provides an opportunity for the infrastructure to transparently handle many kinds of faults.

Service Sockets functionality common to all services (e.g., selecting a server when initiating service, detecting when a peer is no longer available, re-establishing a failed service etc) is provided by a user-level library that can be linked into client applications. In addition, Service Sockets allows service-specific functionality (such as buffering, application-level framing, service-level handoff etc) to be seamlessly integrated as application-level layers on top of the IP stack. To allow different layers to be stacked with relative ease, Service Sockets requires each layer to implement a common interface which we refer to as the *Vsocket* interface (See Figure 6 (b)). Each layer is to be implemented as



(a) structure of the new service to be merged in

(b) structure of the final merged service

Figure 5: Example of a merged service

a dynamically linked library which is transparently loaded whenever it is requested as a part of a Service Socket stack.

The Service Sockets library provides analogues for all standard procedures that manipulate sockets. The Service Sockets versions of these procedures have similar names (`Vsend`, `Vconnect` etc) and the same signatures as the standard procedures (except `Vsocket`). The `Vsocket` procedure takes an additional string argument which specifies the sequence of processing modules to be stacked on top of the IP stack. This argument is a string representation of the module names and the arguments to be passed to these modules (e.g., the key for an encryption module). Descriptions of successive modules are separated by a pipe symbol ("`|`"). Figure 6 (a) contains a schematic example.

3.4.1 The Vsockets interface

The Vsockets interface is a lightweight user-level mechanism to dynamically stack service-specific processing layers on top of the IP stack. The design and implementation of Vsockets is inspired by the *STREAMS* interface used in Unix to layer kernel-level processing modules on top of a character

```
Vsocket("module1(arg=val) | module2 | module3(arg1=v1,arg2=v2)",AF_UNIX,SOCK_DGRAM,0)
```

(a) Schematic of a Service Socket specification

Operation	Description
`dl_init`	invoked from `Vsocket()`, process service-specific parameters
`dl_send`	invoked from `Vsend()`; process/buffer and send data
`dl_recv`	invoked from `Vrecv()`; receive/process and buffer data
`dl_close`	invoked from `Vclose()`, clean up state

(b) The *Vsocket* interface

Figure 6: Illustrations of Service Sockets.

device driver [20, 22]. Service Sockets shares several features with STREAMS. First, they allow processing modules to be chained into a bidirectional data-path, successive modules on a chain being separated by queues. Second, they allow multiplexing to and de-multiplexing from the data-path. Third, data-paths with processing modules chained into them can be polled, blocked on, read from, written to in a manner similar to data-paths with no such modules.

There are, however, several differences between Service Sockets and STREAMS. First, Service Sockets are implemented entirely at the user level and needs no special privileges. This allows Service Sockets to be implemented on any operating system.[4] Second, stacking modules in Service Sockets is cheap enough that applications can consider specifying a different stack for every socket created. This is crucial if the Service Socket abstraction is to be a useful one. Requiring administrator access and/or long installation latencies would render it unusable. Third, the processing modules in Service Sockets are invoked synchronously - the stacked modules don't get to run unless the application makes a call into the Service Sockets library. This allows applications to control the processing sequence for different service sockets. This can be important for applications that access multiple services. Finally, since Service Sockets are implemented as a user-level library, they do not allow mixing data-paths with and without processing modules in a single call to `poll()`. Instead, it defines a separate set of virtual file descriptors which can be used only with functions provided by the Service Sockets library. This is an artifact of the user-level implementation of Service Sockets. While this is no doubt an inconvenience, we believe that it is a reasonable tradeoff for the flexibility and the efficiency provided by a user-level library.

3.4.2 The Service Sockets library

In addition to providing general-purpose support for stacking service-specific layers on top of the IP stack, the Service Sockets library also provides support for initiating and managing Wrens services.

1. If a trailing sequence of modules in a processing chain is not available locally, it is interpreted as a remotely provided Wrens service and is transparently instantiated. Integrating the instantiation of Wrens services with the creation of a Service Sockets processing stack facilitates evolution of services. On resource-rich platforms, simpler Wrens services can be implemented

[4] Versions of the Service Sockets library have been implemented on Solaris, Linux and the PalmOS. We are working on a WindowsCE version.

as a part of the Service Sockets stack; on resource-poor platforms, parts of the Service Sockets stack can be implemented as a Wrens service.

2. The Service Sockets library participates in the Wrens fault detection and recovery protocol (see Section 4.2 for details).

3. Service Sockets allow data being sent to (from) a Service Sockets to be de-multiplexed to (multiplexed from) a set of ordinary sockets – all of which could connected to a single Wrens service component. The mapping between the transport-level connections and a Service Socket can be one-to-one or many-to-one and can be dynamically modified. This ability is useful, for example, for multicast-based services which use multiple related multicast groups: for example, layered multicast [14], self-organizing transcoders [12] and Customcast (see Section 5.4).

3.5 ActiveLDAP

Lookup is the fundamental control construct in Wrens and is used pervasively. Wrens uses lookup to determine the structure of services, identity and suitable location of service components and end-points for legacy links. To support this, Wrens provides *active directories*. Active directories allow a function to be executed as part of the lookup operation. This allows the lookup operation to return different results based on the identity/location of the client, parameters specified as a part of the lookup operation, the previous sequence of lookup requests etc. The dynamic nature of the lookup for an *active directory* allows Wrens to automatically adapt (or compose) a service based on information about server load, client's location and connection to the service infrastructure, client's access device, intended set of peers etc.

To implement active directories, Wrens extends the semantics of the Lightweight Directory Access Protocol (LDAP) [28] to allow a function to be executed as a part of the lookup operation. To ensure compatibility with legacy LDAP-enabled applications, *we did not change the wire protocol*. Instead, the change in the semantics is implemented by introducing a convention for marking LDAP entries as *dynamic* and changing the implementation of the LDAP search operation to additionally invoke a function whenever a *dynamic* name is accessed. We do this by inserting a fixed attribute, `subroutine` in each dynamic LDAP entry. This attribute contains the name of a procedure to be invoked whenever the contents of this node are requested as a part of a LDAP search operation.

We have been able to make introduce this extension to LDAP with very little change to existing implementations. First, we perform a normal LDAP search. Instead of returning the results right away, we check the returned results for nodes with a `subroutine` attribute. If one or more such nodes are found, the corresponding procedures are invoked one by one.[5] Arguments to these procedures can be embedded within the search filter string passed to LDAP. The invoked procedure is expected to return a sequence of `<name=value>` pairs which are returned to the client instead of the contents of the dynamic node. Procedures to be invoked for dynamic nodes are to be implemented in dynamically linked libraries which are loaded as a part of server initialization. We have modified the *SLAPD* implementation of LDAP version 2 [24] for our implementation of *ActiveLDAP*.[6]

We illustrate the structure and operation of our *ActiveLDAP* implementation in Figure 7. The figure shows a directory tree with entries for two Wrens services (`selectcast` and `foo`). There are five dynamic nodes: `get_url`, `set_url`, `get_service_description`, and two others. In the figure, these nodes are marked with an asterisk. In the figure, the *ActiveLDAP* server receives a search

[5]If multiple such nodes are found, the corresponding procedures are invoked in arbitrary order.

[6]We used OpenLDAP [17] as code base for availability and licensing reasons.

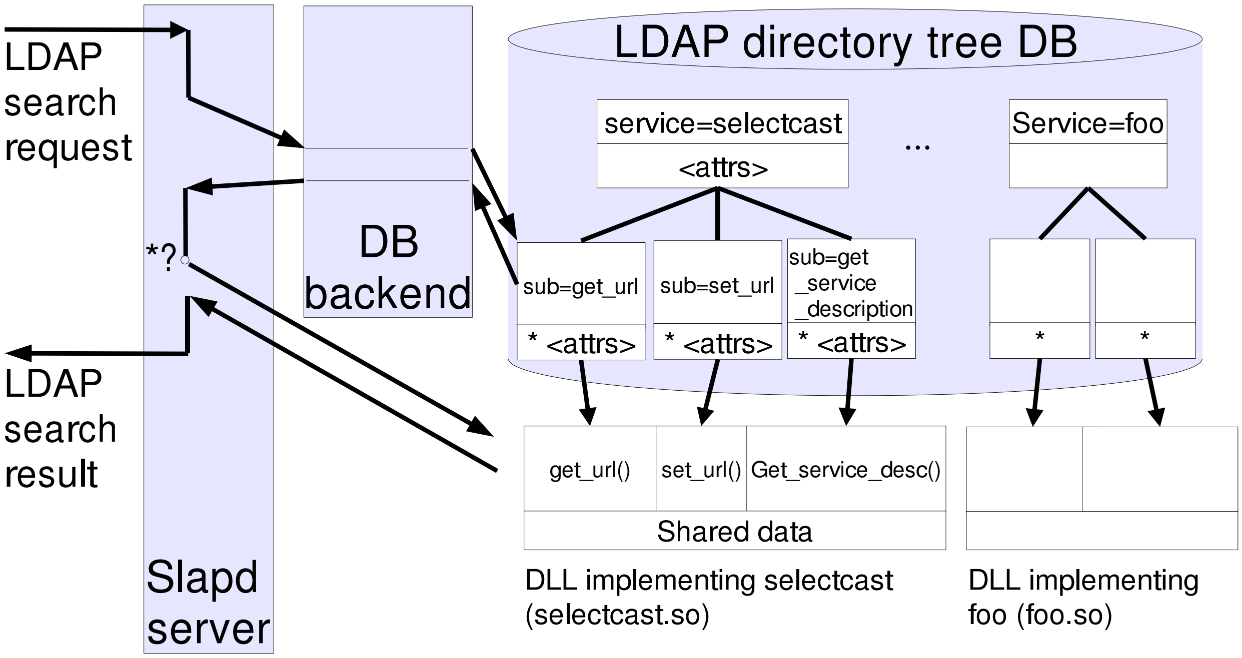


Figure 7: Structure and operation of an *ActiveLDAP* server.

request for a dynamic node `get_url` that is a part of the `selectcast` service (see Section 5.2) The database backend retrieves the node and forwards it to front-end to be returned to the requester. The front-end recognizes the node being returned as being dynamic, and invokes the procedure associated with it (i.e., `get_url`) and returns the result to the requester.

4 End-to-end operation

In this section, we describe the end-to-end operation of the Wrens framework. We focus on two aspects: distributed instantiation of a service with multiple service components and distributed fault-recovery when parts of a service fail.

4.1 Service Instantiation

The Wrench service specification language is fairly powerful and does not restrict the communication topology of Wrens services. To automatically instantiate complex services, care must be taken to order the creation and scheduling of individual service components in way that avoids deadlocks. For services that use TCP connections, all locations participating in service instantiation need to agree on the directionality of each TCP connection (which end `connect()`s and which end `accept()`s). The service component that performs the `connect()` on a connection should be instantiated after its peer that performs the `accept()` on the same connection. Furthermore, the connecting component also needs to know the port number to `connect` to. On a more global scale, the global communication structure has to be taken into account. Given a cyclical service structure, a naive strategy that creates components in the order that they are specified and has every component first try to `accept()` connections from its successor(s) and then `connect()` to its predecessor(s), can result in a deadlock.

<table>
<tr>
<td>
1. Scan incoming service graph to determine the set of components not yet instantiated
2. Try to resolve the locations for all uninstantiated components
3. For each component with a resolved location
(a) update the service graph with resolved locations
(b) if the component is to be instantiated here
i. if service process does not exists, create service process
ii. forward service graph to service process
iii. wait till service process returns an updated graph
(c) else, forward updated service graph to resolved location
</td>
<td>
1. Scan incoming service graph for instances of this component
2. For each instance
(a) create sockets for all streams
(b) for each TCP stream connected to a component later in the instantiation order
i. bind and listen on an ephemeral port. Embed the port number in the service graph
(c) return updated service graph to dispatcher
3. for each TCP stream connected to a component
(a) earlier in instantiation order: <code>connect()</code> to the appropriate (machine, port) pair
(b) later in instantiation order: <code>accept()</code> from ephemeral port
</td>
</tr>
<tr>
<td>(a) <code>dispatcher</code></td>
<td>(b) service process</td>
</tr>
</table>

Figure 8: Operations performed by the `dispatcher` and the service process to instantiate service components.

To determine a safe instantiation order for service components, we perform a topological sort on the service description graph (recall that a Wrench service specification describes a graph with the service components as the nodes and the communication channels between these components as links). The client requesting a service is chosen as root of the topological sort. We use a distributed protocol for instantiating the service, using the result of the topological sort as the instantiation order.

The service description is initially sent to the first location in instantiation order. After receiving a description, each location instantiates all the components that are assigned to it, updates the service description graph to record information about the service components that have been instantiated at this location, and forwards the updated graph to the next location in instantiation order. At each location, the `dispatcher` and the service process cooperate to perform the instantiation. Figure 8 describes the operations performed by each of these. For every TCP connection in the graph, the service component that occurs earlier in the graph `bind()`s a stream socket and `listen()`s on it. It also embeds the port number that it is `listen`ing in the service description graph it forwards to locations instantiated later. The peer, receiving this information, will then complete connection setup by `connect()`ing to the respective port.

4.2 Fault detection and recovery

In this section, we describe the fault detection and recovery protocol used by Wrens. In general, distributed fault-detection and recovery is a hard problem. We believe that while the infrastructure itself cannot possible be expected to solve the general problem of distributed fault-tolerance, support for several common cases can be and should be provided. We therefore focus on a limited version of the problem which is simple enough to be solved efficiently and yet powerful enough to be useful in a variety of scenarios. We assume a fail-stop model for service components, i.e., a service component fails by terminating involuntarily. Failure of a service component can, then, be detected by its peers (by broken connections that cannot be re-established for TCP streams and by prolonged unresponsiveness for non-connection-oriented streams). In the rest of this subsection, we discuss the Wrens fault-detection and recovery schemes. Currently, the fault recovery scheme works as long as only one peer of a service component goes down at a time. This allows Wrens to recover from multiple faults as long as they are separated by at least two nodes in the service graph. If two or more peers of a service component go down, Wrens shuts the service down.

4.2.1 Fault-detection

TCP streams: as mentioned in Section 3.1, every service process consists of a *master* thread and a set of *per-instance* threads. The master thread performs all administrative operations for a service component and the instance threads execute the code for the service component. As a part of its administrative operations, the master thread checks if any of the TCP streams associated with any of the instance threads has been closed. If this happens, and no voluntary-termination message has been received from the corresponding peer (see Section 3.2), the master thread assumes that a fault of some sort has occurred.

Non-connection-oriented streams: for non-connection-oriented streams (UDP/multicast), Wrens relies on the instance threads to detect faults in an application-specific manner (timeouts, periodic heartbeat messages or a service-specific protocol) and to notify the master thread using the `streamFault()` procedure in the stream-based programming interface (see Section 3.2).

4.2.2 Fault Recovery

We describe the distributed fault recovery algorithm as a state machine that is executed independently on every node that detects a faulty peer.

1. **NoFault:** if a TCP connection to a peer is broken, goto **Reconnect**; else, goto **Recreate**.

2. **Reconnect:** for a configurable timeout t_1, try to reconnect (or re-accept) the broken connection. If successful, goto **NoFault**; else goto **Recreate**.

3. **Recreate** the instantiation process establishes an instantiation order based on a topological sort. This defines a unique "parent" for every service component.

   If this process is the parent of the failed peer, try to re-create service component at the same location using the original service graph. If the attempt is successful within a configurable timeout t_2, goto **NoFault**; else, goto **Reinstantiate**.

   If this process is not the parent of the failed peer, wait for the failed peer to signal successful re-creation. If signal received within t_2, reconnect the broken connection (if TCP) and goto **NoFault**; else, goto **Reinstantiate**.

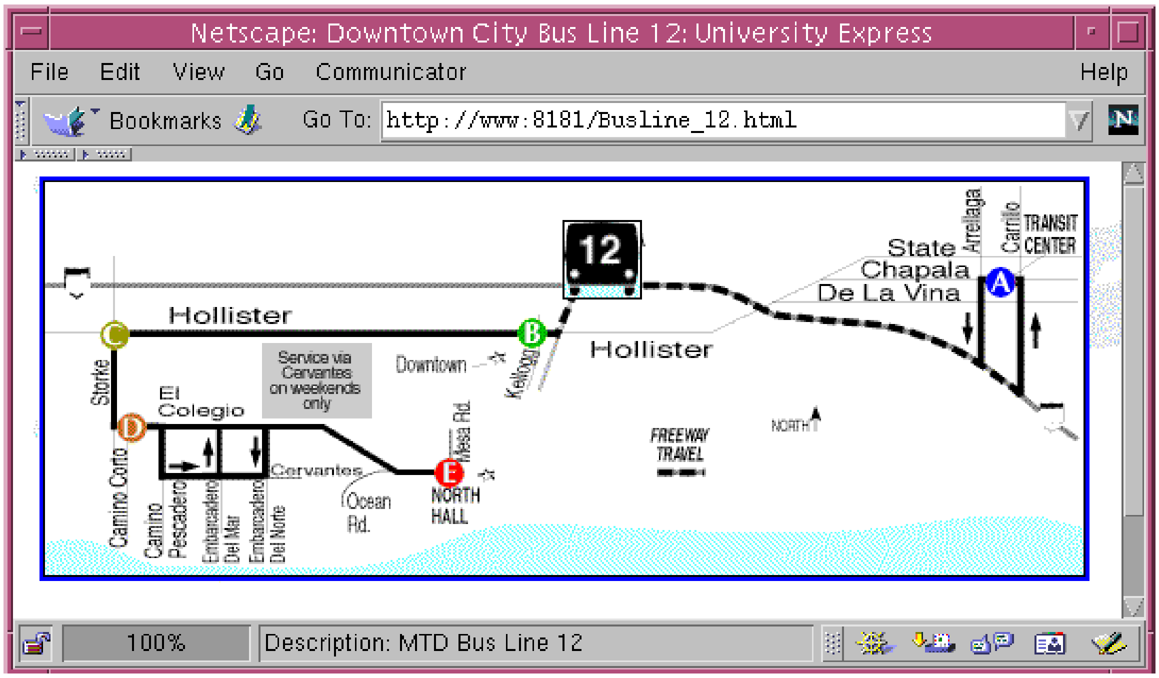


Figure 9: Screen shot illustrating the `porthole` service.

4. **Reinstantiate:** if this process is the parent of the failed peer, and the location of the failed service was defined symbolically, then lookup the location again. If the directory service returns a the location, goto **Shutdown**; else, try for an additional t_3 time units to start the service at the new location. If successful, goto **NoFault**; else goto **Shutdown**.

   If this process is not the parent of the failed peer, wait for the failed peer to signal successful re-creation. If signal received within t_3, reconnect the broken connection (if TCP) and goto **NoFault**; else, goto **Shutdown**.

5. **Shutdown:** abort this service component and notify peers of voluntary termination.

5 Services implemented using Wrens

5.1 The `porthole` location-based information service

Goal of the `porthole` service is to provide mobile clients with up-to-date customized information about geographically localized entities (the view from a `porthole` depends on where you are :)). When clients connect to the `porthole` service, they specify a geographical location and a profile listing the information categories that they are interested in. Examples of information that may be requested include parking lot availability, public transportation schedules *and the current location of buses*, construction zones, accidents, traffic congestion, hotel vacancies, current events, and timely advertisements, e.g., "Lunch at Joe's Kitchen today only $3.95". Figure 9 presents a screen shot illustrating the `porthole` service. It shows a Santa Barbara bus-line information page; the bus icon indicates where the bus was located when the last update was received.

The `porthole` service collects the requested information from different servers and embeds it in a map obtained from a map server specified by the client. It then embeds the map in an HTML file

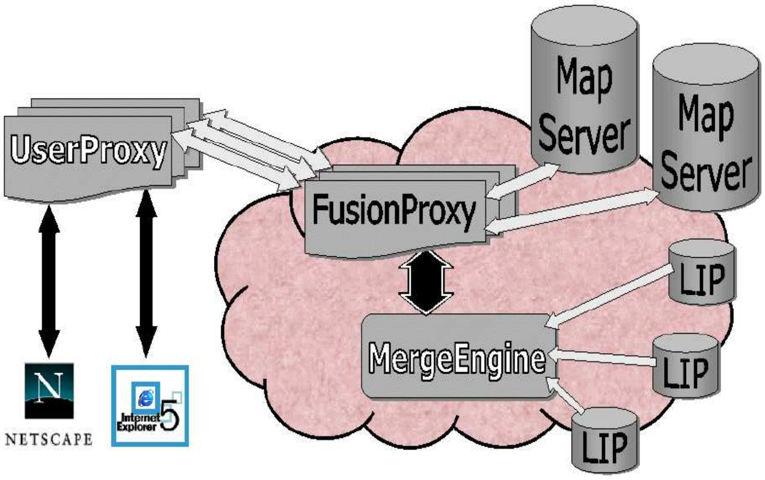


Figure 10: Structure of the `porthole` service

and returns it to the client. The client uses a standard web browser to connect to `porthole` as well as to display the information returned. To facilitate periodic monitoring of the desired information, `porthole` embeds a "reload-after-T-seconds" command in the HTML file.

The `porthole` service consists of three parts: a client-side proxy (the *UserProxy*) and two Wrens service components (the *MergeEngine* and the *FusionProxy*). See Figure 10 for the structure of the `porthole` service. Implementing the `porthole` service took about three weeks.

The UserProxy maintains the geographical coordinates for the machine it is running on (either specified in a profile or obtained from a GPS PCMCIA card). It also keeps an interest profile for the user and allows her to update it using a graphical interface (see Figure 11 for a screen shot). It operates by acting as a web proxy for standard web browsers. It augments the requests it receives with the geographical coordinates and the interest profile of the user (as additional HTTP header fields). It then uses a Service Socket to initiate the `porthole` service.

The FusionProxy strips the geographical coordinates from the request and uses them to construct a query for the map server. The map server returns the map as an image embedded in an HTML file[7]. The FusionProxy passes this information on to the MergeEngine.

The MergeEngine maintains persistent connections to a set of LIPs (local information providers). Each LIP make its information available as an HTML page that contains *IconDescription*s consisting of a geographical location and an icon image. IconDescriptions are indexed by tags that are matched with user interests. Given a user profile, the MergeEngine finds all matching IconDescriptions and inserts the corresponding icon at the correct locations in the map and adds referencing HTML code to the embedding page. It then returns the modified map image and HTML page to the FusionProxy which returns it to the UserProxy.

5.2 The `selectcast` service for sender-defined multicast

The `selectcast` service provides attribute-based messaging. It allows a sender to define the group of intended receivers of a message as a predicate over receiver attributes, e.g., everyone who is running Solaris 5.6 on the second floor of the department, or everyone who is driving a large vehicle north on I-405 and is within fifteen miles of the LAX airport. To avail of this service, receivers

[7] We currently use `http://pubweb.parc.xerox.com/map` as a map server.

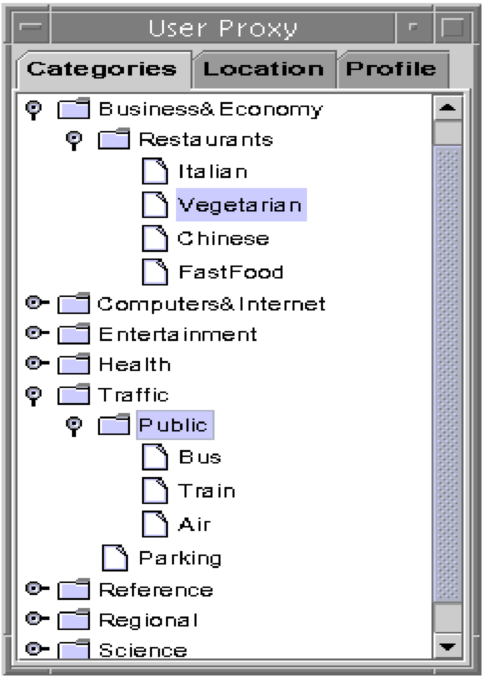


Figure 11: Graphical user interface for the `porthole` UserProxy.

specify a set of attributes on the basis of which they are willing to receive messages (e.g., location, driving direction, platform, etc.). The sender associates a predicate with each message. `selectcast` evaluates this predicate to determine the group of receivers for the message and delivers the message to them. We refer to this predicate as the *delivery predicate* for the message.

The Wrens `selectcast` service consists of three parts (see Figure 12): a dynamically linked library extending the ActiveLDAP server *(Database)*, a Service-Sockets-enabled messaging application *(Sender)* and a Wrens service component which does the actual message delivery *(Proxy)*. The application creates a Service Socket with the `selectcast` service name and the delivery predicate as its parameter. The standard Wrens service setup procedures looks this service up at the active directory (1). The active directory evaluates the delivery predicate and creates and stores a group of receivers whose attributes satisfy the predicate.[8] It assigns an identifier to this receiver group and embeds it in the Wrench service description that it returns to the sender (2). The sender transmits the message using `Vsend` (after forwarding the Wrench description to the proxy as part of the service setup) (3). The proxy has one input and one output stream. The output stream is specified as a *lazy* stream with a symbolic destination that has the identifier of the receiver group embedded *("ldap://selectcast.com:2001/service%msgID=2010")*. Since the output is a lazy stream, every time it is used, the Wrens framework looks up its destination at the ActiveLDAP server (4), (5) and creates a fresh connection (6). The proxy simply forwards the data from its input stream to its output stream multiple times (each time taking steps 4, 5 and 6). The ActiveLDAP server maintains an iterator; every time the destination of this stream is looked up, it returns the next host in the receiver group. When the iterator reaches the end of the receiver group, it frees it. The following lookup with the same symbolic destination returns a lookup failure and the `selectcast` service terminates.

Implementing `selectcast` in the Wrens framework was fairly easy and took less than a day.

[8]The receiver group is stored as *soft-state*. It is reclaimed if no reference is made to it within a given period.

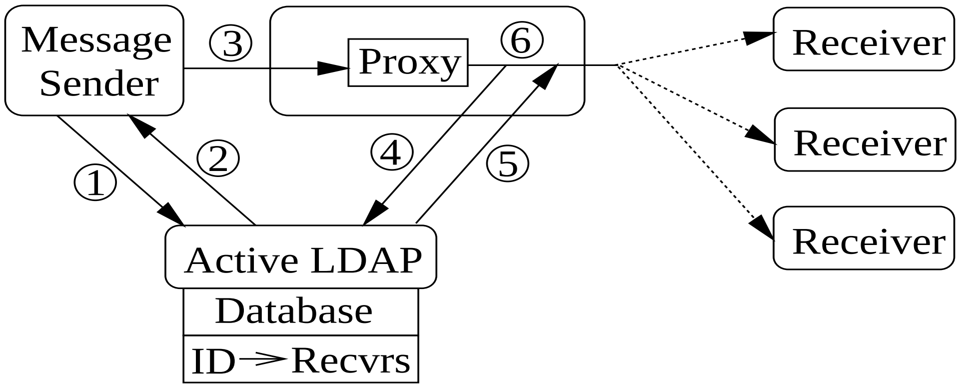


Figure 12: Structure of the `selectcast` service

Most of this time was spent to develop the dynamically linked library for the ActiveLDAP server. The messaging application and the service component used are slightly modified versions of small pieces of code we used for debugging the Wrens framework.

5.3 The `SMP3` service for managing on-demand streaming audio

While we were writing a simple forwarding service component for benchmark purposes, we noticed that streaming MP3 servers (such as *Icecast* [10] and *Shoutcast* [21]) essentially perform the same operation. From this, it was only a small step to extend our forwarding component to allow multiple clients to log in and listen to the same MP3 stream. The extended service component (which is about 200 lines of C code) forms the core of the SMP3 service. It uses *connect* legacy links to connect to existing *icecast* and *shoutcast* servers and *accept* legacy links to accept connections from unmodified MP3 players. Creating the SMP3 service took us a couple of hours. SMP3 is currently used by several of us to listen to music as we work.

In effect, SMP3 is a drop-in replacement for the *Icecast* and *Shoutcast* servers but has several advantages over them. First, it can transparently handle the situation where the source of the MP3 stream resets. The Wrens framework detects this situation and automatically tries to reconnect to the source. Second, it uses a lazy stream to connect to the MP3 source. This allows it to close the source stream if no clients are listening and reopen it as soon as new clients arrive. Third, it was extremely simple to further extend. To convert SMP3 into a multicast-based MP3 broadcasting service required only a few minutes and a small change in the service specification (one of the output streams is specified as a multicast group) with no change in the service code at all. To create a hierarchy of broadcasting nodes required ten minutes of updating the active directory. Figure 13 shows the pseudo-code for the `SMP3mgr` service component. It can be instantiated along the lines of Figure 3.3. Note that the only implicit assumption in the code is that stream 0 is the only source, whose data will be distributed to every client stream. In all other respect, the code itself is unaware of the service topology and can be used with multicast, or unicast (UDP or TCP).

5.4 The `customcast` customized news ticker service

An efficient way of distributing news articles to a large client-base is to group the articles into categories and multicast the articles in each category on a separate multicast group. This technique, however, requires a very large number of multicast groups. The `customcast` service tries to conserve

```
Init(streams) {
   clients = all streams except stream 0;
}
Rebind(streamDiff) {
   if(addOperation(streamDiff))
      clients += streamDiff;
   if(delOperation(streamDiff))
      clients -= streamDiff;
}
Process(instream) {
   if(instream == 0){
      buf = streamRead(0);
      forall c in (clients){
         s = streamFor(c);
         streamWrite(buf,s);
      }
   } /* no data is expected from clients */
}
```

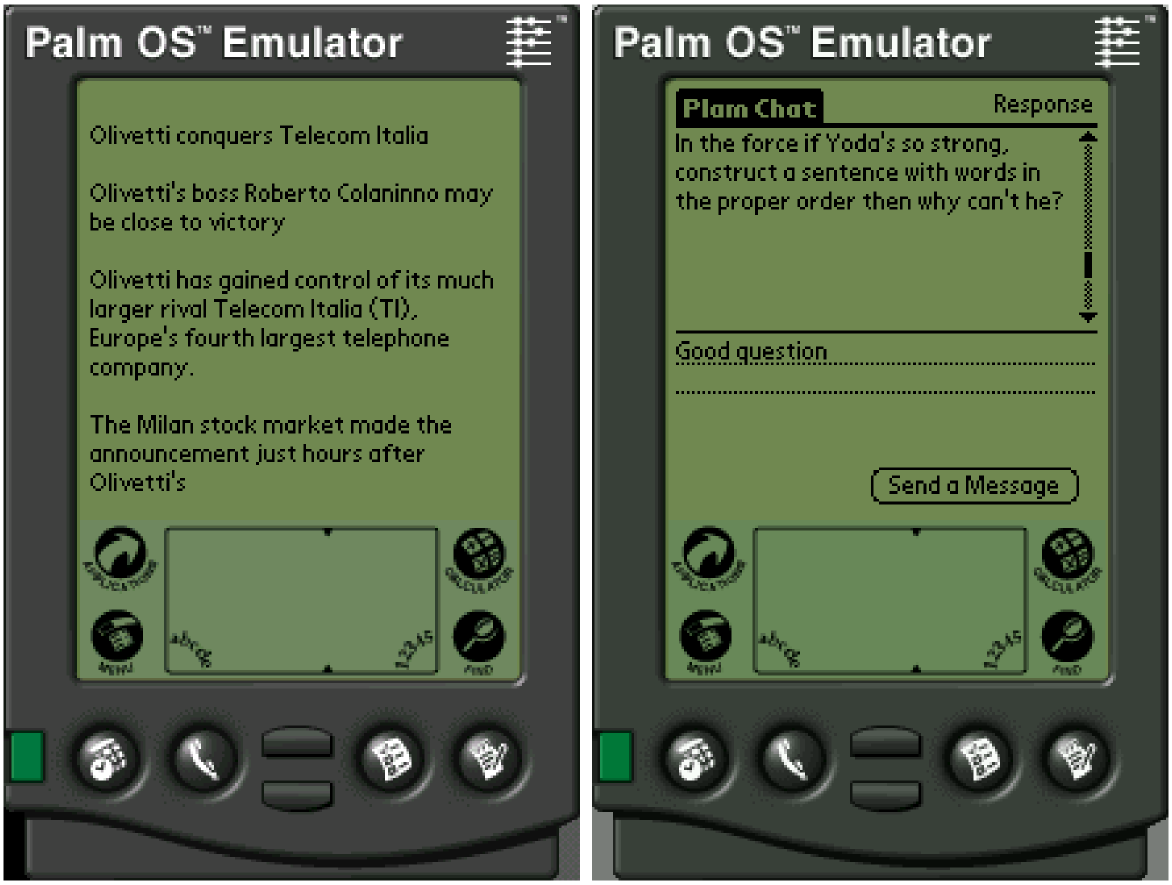
Figure 13: Pseudo-code for the `SMP3mgr` component

multicast addresses by multiplexing some news categories into one multicast group and dynamically changing the mapping based on active user profiles.

The `customcast` service consists of three parts: Perl-based news tickers, a Wrens service component, and a client-side proxy. The news tickers extract articles from a variety of sources (BBC, AP, `slashdot.org`, `freshmeat.org` etc), packetize and categorize individual articles[9] and send them a single stream. The Wrens service component (we refer to it as the Customcast News Node[10]) dynamically maps the news categories onto a set of multicast groups based on the frequency of articles and the number of clients interested in each category. It uses a bin-packing scheme that tries to minimize the number of multicast groups each client has to subscribe to as well as the number of unwanted articles that are delivered to each client. The client-side proxy uses a Service Socket to receive the packets on a dynamically changing set of multicast groups, reassemble articles (discarding articles with one or more packets lost), and filter out unwanted articles. It displays the articles returned by the Service Socket using a standard browser.

This service makes use of the dynamic de-multiplexing capability of Service Sockets to receive data from a dynamically changing set of sockets to a single Service Socket (see Section 3.4). The client-side proxy just opens a single Service Socket and waits in a `Vrecv` loop. The Service Sockets library receives notification from the Customcast News Node about changes in the multicast groups that this client should be listening to and automatically changes the mapping. The remapping is handled by the service-independent part of the Service Sockets library and is transparent to both the client application and the service-specific layers stacked on top of the IP stack.

[9]Currently, we use simple categories based on the source of the articles and a set of keywords.
[10]We had to omit the acronym to avoid trademark problems.



(a) `nuz` service (b) `plamchat`

Figure 14: Screen shots from the PalmPilot services.

5.5 PalmPilot services

The PalmPilot PDA is a widely used hand-held device. It is resource-limited (slow processor, limited memory, few simultaneous network connections, no multicast support and no disk). Hence, our implementation of Service Sockets for the PalmPilot provides only limited functionality. It supports only TCP and UDP and (as yet) has no support for fault recovery. We have used this implementation to develop two services (see Figure 5.5 for screen shots).

The `nuz` transcoding and caching news ticker: The `nuz` service adapts the `customcast` service by adding a store-and-forward proxy between the Customcast News Node and end-user clients. The proxy acts as an adapter that transcodes[11] and stores the last n articles of interest on behalf of a registered (but disconnected) mobile client. Mobile clients connect to the proxy intermittently to download the buffered data. The interest profile as well as the set of messages for a client are stored as *soft-state* and are deleted if not retrieved in a given time period. The development of this service also demonstrated how new components can be easily plugged into existing Wrens services – it took us less than a day to implement the `nuz` service.

[11]The store-and-forward proxy transcodes incoming data to the PalmPilot doc format.

The `plamchat` secure chat service: to implement a multi-client chat service, we modified the forwarding service component from the `SMP3` service. Instead of forwarding the data from only one designated source stream, the modified component forwards data from any stream on to any other. Coding this change, removing the HTTP-header parsing code used by `SMP3`, installing the chat service, and exchanging the first messages took about 45 minutes. This chat service can be used by clients using PalmPilots or any telnet session. Later, we used the Secure Socket Layer (SSL) library to provide encryption at the PalmPilot and inserted a simple decryption proxy between the PalmPilot and the chat service.

6 Performance and overheads

The main goal of Wrens is to facilitate the rapid evolution of network services. While performance is not an explicit goal, it is important to ensure that acceptable performance can be achieved. In this section, we present an evaluation of the overheads introduced by the Wrens framework and the costs of end-to-end operations.

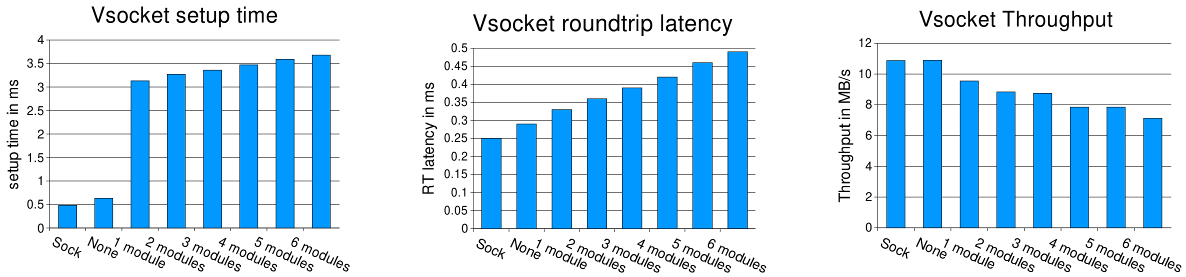
We performed three sets of experiments. The first set of experiments measured the overhead introduced by Service Sockets. The second set of experiments measured the cost of end-to-end operations in the Wrens framework. We performed these experiments in both a LAN setting and a wide-area setting. For these experiments, we used the connection setup time, the round-trip latency for a single byte and the throughput for large messages as the metrics of performance. The third set of experiments measured the time taken by the Wrens fault detection and recovery mechanism to repair services of different sizes and topologies. These experiments were performed on a cluster of Sun Ultra-1, Ultra-30, and Ultra-450 machines, all running Solaris 2.6 and connected by fast Ethernet. All experiments were repeated until the measurements established a 90% confidence interval of being within 5% of the mean.

6.1 Overhead due to Service Sockets

To determine the overheads introduced by Service Sockets, we compare the performance of three configurations. The first configuration uses standard sockets to connect two programs, the second configuration uses Service Sockets without any service-specific layers on the stack. The third configuration uses Service Sockets with a varying number of simple forwarding layers. In this last configuration, both peers use the same number of service-specific layers, and all but the bottom modules in the stack perform local copying and buffering.

Figure 15 presents all three metrics (connection setup time, round-trip latency, throughput) for all three configurations. We note that for all three metrics, performance of Service Sockets with no layers stacked is close to that of standard sockets. This shows that the inherent overhead of the Service Socket library is very small. The connection setup time for Service Sockets increases sharply when the first layer is added to the stack and slowly thereafter. By inserting timing probes in the Service Sockets library, we determined that almost all the additional cost for adding a new stack layer to a Service Socket is due to the dynamic linking operation to load the code for the layer.

The round-trip latency for Service Sockets increases linearly with the number of stacked layers. Each additional layer adds 0.03 ms to the round-trip latency – which is the cost of invoking each layer on the Service Sockets stack. The throughput decreases roughly linearly with the number of



(a) connection setup (b) round-trip latency (c) throughput

Figure 15: Overhead introduced by Service Sockets.

stacked layer.[12] This is to be expected as each layer copies data from an input buffer to an output buffer.

6.2 Cost of end-to-end operations

The cost of setting up a multi-component service depends on the topology of the service and the number of components. To evaluate the impact of the topology on the cost of setting up a service, we ran two sets of experiments: one with a chain topology and the other with a clique (i.e., fully-connected or all-to-all) topology. We selected these topologies as they represent the extremes in the terms of the number of peers for every component. For each topology, we varied the number of (identical) components from one to eight. We used a simple service component which forwards data from any input streams to all output streams. A part of the cost of setting up a Wrens service is due to the additional messages that are exchanged between peers to maintain information needed for fault recovery. To determine the overhead introduced by these messages, we repeated both the experiments with the fault recovery mechanism disabled. To determine the impact of Service Sockets for services provided over long-haul links, we repeated these experiments with machines at three geographically distant locations – University of California, Santa Barbara (on the US west coast), Carnegie Mellon University (on the US east coast)[13] and the Technical University at Munich (Germany).

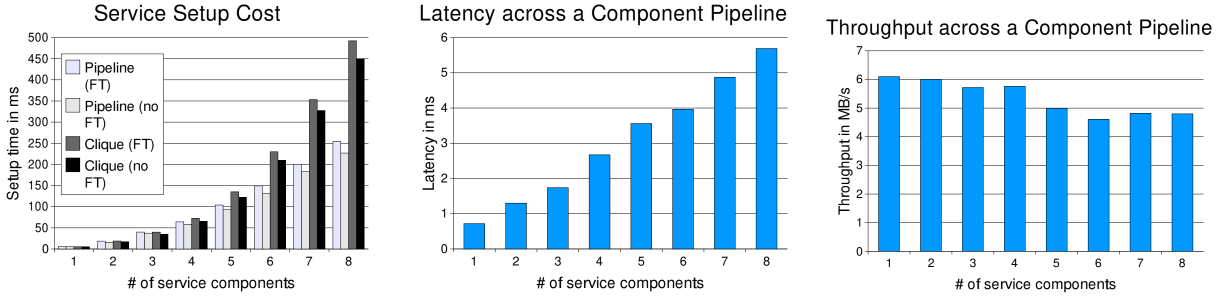
6.2.1 Results for a LAN scenario

The results for all four sets of experiments are presented in Figure 16 (a). We note that the overhead introduced by the fault recovery mechanism is less than 10% for all the configurations tested. This is particularly encouraging since in the absence of faults, the fault recovery mechanism does not incur any other costs. We further note that for small configurations (less than four components), there is little difference between the setup time for services with different topologies. For larger configurations, the difference in the setup time between the chain and the clique topologies increases with the number of components. To put these results in context, we would like to point out that all the services that we have described in Section 5 have less than four components.

To measure the end-to-end round-trip latency and throughput in the Wrens framework, we ran experiments using a chain topology with a varying number of components. The results are

[12]Note that the decrease is only roughly linear. As yet, we have not been able to explain the variations from a linear decrease seen with three and five layers.

[13]CMU is "almost" on the east coast :)



(a) connection setup (b) round-trip latency (c) throughput

Figure 16: Cost of end-to-end operations in the Wrens framework (LAN scenario).

TCP connect()	UCSB-CMU	161ms
TCP connect()	CMU-TUM	245ms
TCP connect()	TUM-UCSB	412ms
Wrens setup	UCSB-CMU	416ms
Wrens setup	UCSB-CMU-TUM	981ms
ICMP ping	UCSB-CMU	158ms
ICMP ping	CMU-TUM	240ms
Wrens RT latency	UCSB-CMU	159ms
Wrens RT latency	UCSB-CMU-TUM	415ms
TCP send()	UCSB-CMU	88kB/s
TCP send()	UCSB-CMU-TUM	36kB/s
Wrens throughput	UCSB-CMU	83kB/s
Wrens throughput	UCSB-CMU-TUM	33kB/s

Table 1: Network measurements for connections from our site (UCSB) to CMU and the TU Munich (TUM).

presented in Figures 16 (b) and (c) and show that, as expected, the round-trip latency increases linearly (about 0.5 ms per component) and the throughput decreases roughly linearly as the number of components increases.

6.2.2 Results for a wide-area scenario

To understand the overhead introduced by the Wrens framework for end-to-end services in a wide-area setting, we measured the setup time, latency and throughput for 2-component and 3-component services with a clique topology. We measured the time to instantiation time, round-trip latency and throughput for 2-component and 3-component fully-connected service graphs. To put these results in context, we compared them with the time for a TCP `connect()`, an ICMP `ping` and the plain TCP bandwidth using a simple forwarding agent written in C.

The results presented in Table 1 show that the overhead incurred by Wrens is small compared to the delays and bandwidths we find in a typical WAN setting

1. The Wrens setup time of 416ms for a 2-component service compares well with the raw time of 161ms + 245ms= 406ms just needed for the necessary calls to `connect`, ignoring all other

overhead (memory management or calls to `accept()` which a hand-coded service would also have to deal with. Similarly, the 981ms for the Wrens 3-clique setup compares favorably with 161ms +245ms + 412ms= 818ms.

2. The Wrens round-trip latency between UCSB and CMU is 159ms which is very close to the ICMP round-trip latency; the Wrens round-trip latency for the two-hop UCSB-CMU-TUM setup is 415ms which is only about 4% (17ms/415ms) slower than the sum of the component round-trip latencies (158ms+240ms=398ms) Note that the round-trip time cannot be expected to better than the ICMP round-trip latency.

3. The Wrens throughput for the UCSB-CMU setup is about 5% smaller than plain TCP throughput; the Wrens throughput for the UCSB-CMU-TUM setup is about 8% smaller. We would like to note that the results of the wide-area throughput experiments should be viewed with caution as the variation in the numbers was large. The results reported here are the best achieved for each alternative. The results for all other experiments, however, report means with tight confidence intervals.

6.3 Performance of fault recovery

To study the performance of the fault recovery mechanism, we performed two sets of experiments. For the first set of experiments, we determined the time to repair multi-component services if one of the *service components* crashes. For the second set, we determined the time to repair multi-component services if the *dispatcher* on one of the nodes participating in the service crashes.

For the first set of experiments, we used three service configurations: a 3-component chain, an 8-component chain and an 8-component clique. All service components used in these experiments were identical and forwarded all data on their input stream(s) to their output stream(s). Our goal in selecting these configurations was to study the impact of the topology as well as configuration size. For each configuration, we killed one of the intermediate service components and measured the time it takes for end-to-end service to be restored. Restoring end-to-end service took about 60 ms for the 3-component chain, about 92 ms for the 8-component chain and about 135 ms for the 8-component clique. These results are encouraging particularly for the larger configurations (for comparison, see the end-to-end service setup times in Figure 16 (a)).

For the second set of experiments, we used a 3-component chain and killed both the `dispatcher` and the service process corresponding to the middle component. In this case, restoring end-to-end service took 1549 ms. This time can be broken down into 1000 ms (two 500 ms heartbeats) needed for the `watchdog` process on the node to detect that the `dispatcher` is dead and to restart it; 500 ms back-off time for the parent of the failed component to try to recreate it and 50 ms for the book-keeping.

7 Related work

Several researchers have proposed cluster-based infrastructures for network services [1, 6, 7]. Amir et al [1] propose an cluster-based architecture for hosting real-time transcoding services for continuous media, Fox et al [6] have proposed the TACC model for transformation, aggregation, caching and customization services. Gribble et al [7] propose *Ninja*, a Java-based infrastructure that uses mobile code to deploy services and dynamically generated code to introduce runtime-generated levels of indirection.

Of these, Wrens is closest in spirit to *Ninja*. We share with the *Ninja* project, a vision of easy-to-write network services and the principle of introducing a level of indirection between the service provider and the service consumer to gain flexibility. However, we differ in several important ways. First, we do not limit ourselves to a single cluster. Wrens provides support for distributed instantiation and fault recovery for complex services spanning several platforms. Second, instead of burying the communication structure of services in the code for individual modules (or runtime-generated redirector stubs), we propose that the communication structure of the services should be specified separately in a form that can be understood and used by the service infrastructure. Specifying the communication structure of a service in this manner instead of burying it in the code for individual components allows service designers to rapidly develop or extend services by plugging together service components. Third, we propose a stream-based programming model for service components instead of the RMI-based model proposed by *Ninja*. Service modules written in a stream-based model are usually easier to plug together and to reuse as the interactions between such modules are limited to explicitly named channels.

Another major thrust for building network services is based on distributed object systems such as CORBA [8] and Jini [25]. The key idea behind this approach is to develop service components with well-defined interfaces instead of monolithic services. These components can discover each other using a ubiquitous lookup infrastructure and can interact with each other to provide the higher-level services desired. Since the components have well-defined interfaces which can be dynamically discovered, it is expected that it would be easier to develop new services by combining service components. We agree with this approach on several points, in particular, on the advantages of decomposing monolithic services into reusable service components with well-defined interfaces. However, we believe that it is important to explicitly specify the interactions between different components instead of burying them in the code for individual participants. An explicit specification of the interactions makes it easier to understand the behavior of an existing high-level service and to develop new ones.

Several researchers have proposed the use of *Active Networks* to deploying service code within the network [9, 23, 26]. The key idea is to extend packet formats to include code as well as payload data and to extend network routers to execute code embedded in packets. While this approach is extremely general, it is probably more suitable for evolving low-level network protocols instead of high-level services.

Finally, we would like to mention the *path* abstraction introduced by the Scout [15] operating system. The key idea behind this abstraction is to make the interactions between a sequence of processing modules explicit. This enables optimization of processing along a path as well as integrated scheduling of paths. We share with them the idea of making interactions between components explicitly. As mentioned earlier, explicit specification allows Wrens to automatically deploy services, to transparently detect when a service component fails and to repair the service by restarting the component.

8 Conclusions and future work

In this paper, we have presented the *Wrens* framework for rapidly evolvable network services and have described our experiences with developing and deploying a diverse suite of services. Our experience has been extremely positive. We have been able to create a quite a few useful services in a surprisingly short time. More interestingly, for most of these services, we were able to easily and rapidly reuse or adapt existing service components. For instance, the the simple TCP forwarding component we used for our debugging our implementation was the basis for a streaming MP3 server which, in turn, was the basis for a chat-server. We were also able to convert our streaming MP3 server from a single unicast-based server to a multicast-based broadcasting server and, thence, to a hierarchy of multicast-based servers just by making small changes in the service specification and without modifying any code or recompiling.

Our evaluation of the overheads and costs associated with Wrens shows this flexibility and ease of evolution does not come with a large performance price tag. Our experiments show that for moderately sized services (four components or less), the time taken to setup a Wrens service is less than the round-trip ping latency across the country (which is currently about 70 ms). Routing a message through intermediate service components had only a small impact on the latency and throughput in our experiments. We would, however, like to caution the reader that these experiments were performed on lightly loaded machines.

In this research, we have focused on the ease of developing and deploying network services. We believe that we have been successful in developing a framework that makes these tasks easier and that our framework is directly usable for a fairly large class of information providing services. However, there are several open problems that we have not addressed including fault recovery for stateful services, user authentication and access control and providing a user-requested quality of service. In future work, we would like to address some of those issues, in particular the problems of fault recovery for stateful services and user authentication. Finally, we are hoping to develop a variety of new services for mobile and wireless devices.

References

[1] E. Amir, S. McCanne, and R. Katz. An active service framework and its application to real-time multimedia transcoding. In *Proc. of SIGCOMM'98*, pages 178–89, 1998.

[2] W. Auckerman. Japanese mobile phone users get internet access. Internet News Article[14], Feb 1999.

[3] Bluetooth: Main Page. *http://www.bluetooth.com/v2/default.asp*.

[4] The Clarion AutoPC. *http://www.autopc.com*.

[5] D. Farber. Predicting the unpredictable. Distinguished Lecture, Dept of Computer Science, University of California, Santa Barbara, Mar 1999.

[6] A. Fox, S. Gribble, Y. Chawathe, E. Brewer, and P. Gautheir. Cluster-based network services. In *Proc. of SOSP'97*, pages 78–91, 1997.

[7] S. Gribble, M. Welsh, E. Brewer, and D. Culler. The multispace: an evolutionary platform for infrastructural services. In *Proc. of the 1999 USENIX Annual Technical Conference*, 1999.

[8] Object Management Group. *The Common Object Request Broker: Architecture and Specification*. OMG, Inc., 7 1995. Rev. 96-08-04.

[9] M. Hicks, P. Kakkar, J. Moore, C. Gunter, and S. Nettles. PLAN: A packet language for active networks. In *Proc. of ICFP'98*, 1998.

[10] Icecast: an MP3 internet broadcasting system. *http://icecast.linuxpower.org*.

[14] Available at *http://www.internetnews.com/intl-news/article/0,1087,archive_6_74161,00.html*

[11] Intel Connected Car PC Technology - Third Party Technology. *http://www.intel.com/-technology/carpc/tech/index.htm*, 1999.

[12] I. Kouvelas, V. Hardman, and J Crowcroft. Network adaptive continuous-media applications through self organised transcoding. In *Proc. of NOSSDAV'98*, 1998.

[13] J. Lewell. Motorola promises internet browsing on entire mobile phone range. InternetNews Article[15], March 1999.

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

[15] D. Mosberger and L. Peterson. Making paths explicit in the Scout operating system. In *Proc. of OSDI'96*, pages 153–68, 1996.

[16] Nokia 9000il Phone Page. *http://www.shopnokia.com/beautylandscape/0,1004,14,FF.html*.

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

[18] G. Papadopoulos. Billions & Billions of Devices Served: Convergence in the Internet Age. Distinguished Lecture, College of Engineering, University of California, Santa Barbara, February 1998.

[19] J. Paradiso. Mobile Devices. PLDI'99 Panel on Trends in the Future of Industrial Computing, May 1999.

[20] D. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, 1984.

[21] The Shoutcast distributed streaming audio system. *http://www.shoutcast.com*.

[22] Sun Microsystems Inc. *STREAMS Programming Guide*, Jan 1997.

[23] D.L. Tennenhouse, J.M. Smith, W.D. Sincoskie, D.J. Wetherall, and G.J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, January 1997.

[24] University of Michigan LDAP Development Team. *The SLAPD and SLURPD Administrator's Guide*, 1996. Available off *http://www.openldap.org*.

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

[26] D. Wetherall, J. Guttag, and D. Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols. In *Proc. of IEEE OPENARCH'98*, 1998.

[27] Who Makes AutoPCs? *http://www.microsoft.com/windowsce/Products/apc/manufacturers.asp*, April 1999.

[28] W. Yeong, T. Howes, and S. Kille. Lightweight Directory Access Protocol. Network Working Group, RFC 1777, Mar 1995.

[15] Available at *http://uk.internet.com/uk-news/article/0,1087,archive_101_82681,00.html*