FARADs Prototype Design Document

Venkata K. Pingali   Aaron Falk
Theodore Faber   Robert Braden
Information Sciences Institute
University of Southern California
Los Angeles, CA 90292
{pingali,falk,faber,braden}@isi.edu

June 11, 2003

Abstract

This document describes the design of the FARADs prototype in detail. The objective of the prototype was two fold. The first objective was to give a form to the ideas presented in original FARADs architecture document by David Clark [Cla02], and build a platform for experimentation. The second objective was to verify that we indeed get the flexibility and extensibility that was claimed as a benefit of the design. We made several design decisions while instantiating the various components of the architecture primarily due to feasibility and time constraints. We discuss the design in detail, and identify potential future work. The appendix contains the FARADS Programmer's Manual that discusses the various interfaces, and data structures in more detail.

1 Introduction

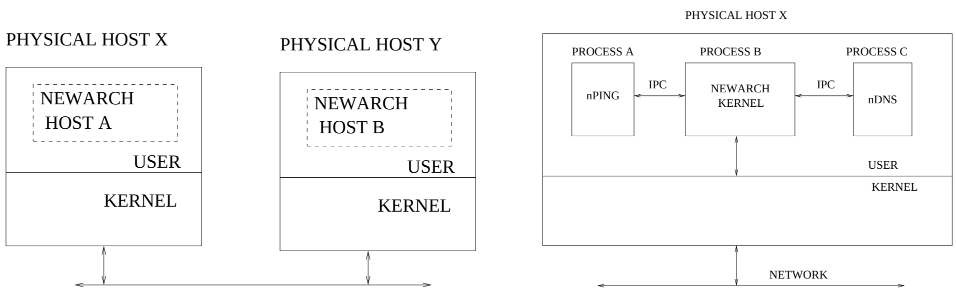
This document describes the overall design of the FARADS prototype and how the various FARADs concepts map to the prototype components. This document assumes that the reader is familiar with the FARADs concepts that are described in the vision [1] and high-level design[2] documents. The purpose of the effort is two fold. The first purpose was to create a platform for testing ideas related to FARADS concepts. The second purpose was to verify that implementing FARADS does indeed lead to flexibility and extensibility of the Internet. Both these objectives were met to varying degrees. By supporting a variety of forwarding and transport mechanisms we show that the transport layer can evolve independently of the forwarding mechanisms. By supporting newer forms of mobility, and as a side effect elimination of some of the issues in existing internet such as NAT support, we show that the FARADs design provides new capabilities.

The prototype consists of group of co-operating processes spread out in the network. UNIX processes and IPC are used to create an application overlay that is the FARADs network of hosts and routers. The prototype also supports some simple applications such as an http application.

The outline of the rest of the document is as follows. Section 2 discusses the overall structure of the prototype. Section 3 presents an example sequence of operations for a mobile entity and a fixed entity. Section 4 lists the various design decisions made while building the prototype. Section 5 describes the control and data flow in the prototype. Section 6 describes support for mobility in FARADs using agents. Section 7 discusses the FD translation problem, and presents canonical routes as a potential solution. Section 8 describes the various protocols providing different communication abstractions. Section 9 discusses the packet formats in more detail and present the various FD formats supported. Section 11 presents two initial scenarios used for testing FARADs. Section 12 concludes by presenting some future work that can build upon the prototype.

2 Prototype Structure

The FARADS network consists of a set of interacting FARADS hosts and routers. In the prototype, each new FARADS host is simulated using a set of UNIX processes as shown in Figure 1(b). Although theoretically multiple FARADS hosts can be simulated on a single host, our current implementation allows only one FARADS host per physical host. These simulated FARADS hosts interact to form an overlay network as shown in Figure 1(a).



(a) FARADS network

(b) FARADS host structure in more detail

Figure 1: Abstract model of FARADS network

Element	Implementation	Function
Entity	Unix process	Application
FARADS Kernel	Unix process	forwarding, local resource discovery
Agent	Unix process	Mobility support
Association	abstract datastructure	Maintain communication state
FD	Multiple forms (IPv4, SlotID), (IPv4 Addr 1, IPv4 Addr 2,..., SlotID n)	Carry routing instructions to deliver packet to the destination entity
Rendezvous	Association establishment protocol	Setup association state at endpoints
Rendezvous String	String carrying protocol cookies	Carry additional first packet information
Slot	abstract datastructure built on Unix socket	Attachment point for the entity
Portal	abstract datastructure	Attachment point for an association
FD Mgmt	Distributed among association and kernel	Perform functions that require knowledge from below-the-line e.g., FD manipulation

Table 1: FARADs design-prototype mapping

We use the phrases "a FARADS host" and "a cluster of processes simulating the FARADS host" interchangeably. The FARADS hosts communicate using a standard protocol such as UDP through the socket interface provided by the operating system. This socket to which the FARADS host binds to effectively acts as the network interface to the FARADS host, and the bidirectional channel created acts as a physical layer. FARADS hosts are multi-homed when the correspoding physical host is multi-homed, and such hosts act as FARADS routers.

Each FARADS host may contain many entities as shown in Figure 1(b), each of which may contain many associations. The FARADS host is built using two components – a FARADs kernel daemon that supports all below-the-line functionality, and an entity process that contains all above-the-line functionality. There is one process corresponding to each entity. The kernel daemon is an essential part of the host, and a minimal host has only the kernel daemon running. Processes corresponding to entities are spawned as and when required. This structure decouples the development of the kernel daemon from that of the entities. All entities communicate with each other and with other entities on the network through the daemon. For this purpose, each entity maintains a communication channel with the daemon using UNIX IPC facilities.

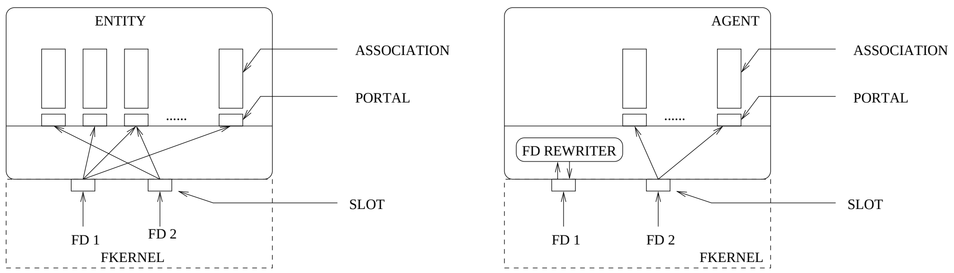
Table 1 lists the various elements of the FARADs architecture, what they map to in the prototype, and gives a summary of their function. In the rest of this section, we explain Table 1 more detail.

2.1 Entities

Figure 2(a) shows an entity containing many associations and two slots, reachable using FD1 and FD2. The entity communicates with the kernel and other entities through the slot. In the prototype, each entity maps to one process. We refer to process and entity interchangeably. At startup time, an entity/process establishes communication channel with the FARADS kernel (daemon), and requests a slot. Once allocated, the entity queries the routing subsystem for an FD corresponding to the slot allocated, called *myfd*. Upon obtaining *myfd*, the entity is ready to communicate with other entities in the FARADS network. In server mode, a non-mobile entity probes the FARADS kernel's local resource discovery function for the FARADS DNS's FD. The entity then registers a name and *myfd* with the DNS. A mobile entity, on the other hand, first probes the kernel for an agent FD. The mobile entity then registers with the agent, which returns a globally routable FD that can be advertised through the DNS. The mobile entity then registers a name and the agent-returned FD with the DNS just like the non-mobile entity.

2.2 FARADS Kernel

The FARADs kernel is an essential component of the FARADS host that is responsible for forwarding packets. It runs as a daemon in the background. The daemon has two interfaces. On the FARADS



(a) Structure of an entity (b) Structure of an agent

Figure 2: Generalized entity and agent structure

network interface, it listens for communication from other FARADS hosts. On the local interface, it listens to all the open IPC channels to entities for control and data messages. The kernel routes messages from the network interface to the local interface, and vice-versa.

For sending and receiving packet on the network interface, we experiment with two possible implementations with different levels of flexibility. In the first implementation, IP packets are constructed by the daemon and sent out of the physical network host bypassing the host operating system such as FreeBSD. In the second implementation, and FARADS host communicate using UDP tunnel. The prototype implementation is single threaded, and supports IPv4 and IPv6 as the link layer protocols.

The slot interface supports four system calls, i.e., send, receive, slot allocate and slot deallocate. The send and receive are equivalent to the traditional UNIX system calls *send* and *recv* provided to the applications. An outgoing packet processing is identical to that of the routed packet. The slot allocate and deallocate are handled by a special control control function that manipulates the mapping between the slot numbers and appropriate OS construct such as the named pipes corresponding to each entity. Section 5 discusses the control flow in more detail.

2.3 Agent

Agent is a special kind of entity that has a globally routable FD and helps other entities establish contract with mobile entities. This is done through rewriting headers before forwarding the messages or sending a redirection message to the source. Its functionality is below the kernel-entity boundary. It must also communicate with the mobile entity to ensure that the latest FD is used forwarding. The implementation uses a modified entity to perform the functions as shown in Figure 2(b). An entity-level implementation was done due to time constraints and since it does not result in any loss of generality.

2.4 Slots

Slots are implemented as abstract datastructures that user unix sockets to communicate with the entities. An entity bootstraps by instantiating the abstraction, and then requesting the kernel to assign the instantiation a specific slot number. The slot state consists of operating system file descriptors corresponding to the sockets and message sequence numbers. Slot state is required on both sides of the kernel-entity boundary. The current slot implementation uses only one socket connection, and does have any staging buffers. A different implementation of the kernel-entity communication channel can have one or both of them. Also, there is no checkpointing or any other kind of support for mobility within the slots.

2.5 Association

Association in the prototype take the form of a abstract datastructure. An entity instantiates an association object to initiate communication. A simple handshake protocol between two instantiated association objects from different entities is used to create an end-to-end association. The association state consists of the source and destination forwarding descriptors, source and destinations AIDs, send and receive buffers, agent FD corresponding to the remote end, and handshake protocol state. Section 8 discusses the various types of associations supported by the prototype in more detail.

2.6 Portal

A portal is an abstract point of attachment for the association. There is no data structure associated with the portal in the current implementation, but in a future implementation involving a complex association, the portal might be used as a staging location before messages are delivered to the association.

2.7 FD Management

The FD Management module consists of the set of functions that require application-level input and below-the-line information simultaneously to perform a desired operation. These include FD manipulation based on user inputs, and detection of changes in FDs used to communicate with a specific entity. The prototype supports only FD change detection. Full fledged mobility support will require a more complex FD management unit operations such as slot allocation at a remote node where the entity plans to move.

3 Operation

A typical operation sequence for a mobile server entity is as follows:

1. The entity obtains a *myfd*, an FD to itself, by querying the routing subsystem.
2. The entity (process) obtains an Agent FD. This FD is either preconfigured/well known or the entity queries the FARADs kernel.
3. This agent FD is presented to the FD management layer along with any route preferences that the entity may have. The FD management layer returns a modified FD.
4. A standard agent registration protocol is used to register a service name to *myfd* mapping. A full fledged association with the agent may or may not be necessary for the registration process.
5. The association, if any, is then terminated.
6. The entity (process) then obtains the FARADS DNS FD. This FD is either preconfigured/well known or the entity queries the FARADs kernel.
7. This DNS FD is presented to the FD management layer along with any route preferences that the entity may have. The FD management layer returns another FD.
8. This new FD is used to establish an association with the DNS.
9. A standard DNS registration protocol is used to register a service name to *myfd* mapping.
10. The association is then terminated.

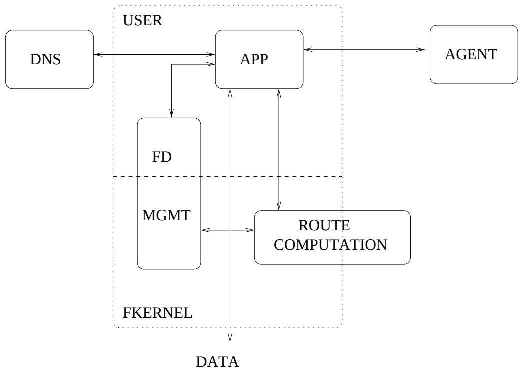


Figure 3: Modules

11. The entity then listens on *myfd* for any new associations.

A fixed server entity can skip the agent registration process. An operation sequence for the client entity is as follows:

1. The entity obtains a *myfd*, an FD to itself, by querying the routing subsystem.

2. The entity (process) then obtains the FARADS DNS FD. This FD is either preconfigured/well known or the entity queries the FARADs kernel.

3. This DNS FD is presented to the FD management layer along with any route preferences that the entity may have. The FD management layer returns another FD.

4. This new FD is used to establish an association with the DNS.

5. A standard DNS querying protocol is used to lookup a service name.

6. The association is then terminated after receiving server's FD.

7. The entity presents server's FD to FD management layer along with routing preferences.

8. The FD management layer returns a modified FD which is used to establish an association with the server entity.

The entity has interfaces to the various other modules such as the routing subsystem, FD management, kernel, agent and the DNS. Only the interfaces relevant to the prototype have been identified in some detail. For other interfaces such as that between the FD management and the routing subsystem, they are part of the future work.

4 Design Choices

The nature of the prototype allowed for several possible implementations. Here we discuss some of our design decisions.

1. User-level implementation. The user-level implementation allowed for fast prototyping. Further, our focus is flexibility and functionality and not performance, which is usually the primary reason for a kernel-based implementation.

2. FreeBSD as the experimental platform. The choice of FreeBSD was fairly arbitrary and most of the code is portable to other unix platforms.

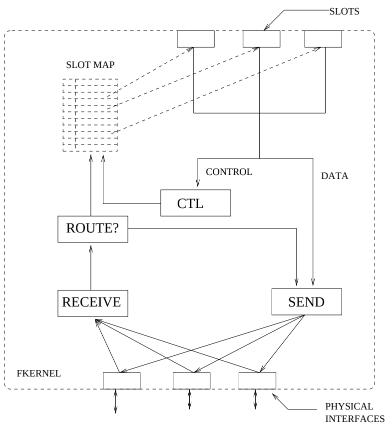


Figure 4: Control and data flow in the FARADS kernel.

3. Use of IP as link layer. A hybrid implementation is possible in which the FDs are expressed in terms of the IP header fields and initial few bytes of the payload. An initial implementation was based on this idea. However, to allow for experimentation with different types of FDs and addressing schemes that are incompatible with IP, IP is only used as the link layer.

4. Kernel and entities as processes. This allows for independent development of the kernel and the individual applications.

5. UNIX IPC for kernel-entity communication. UNIX IPC was used for this purpose because of time constraints, low complexity with reasonable performance. However, the IPC usage requires that the NewArch kernel and entities lie on the same physical host.

6. Agent as a special entity. Although the agent functionality is below the line, a simple but inefficient user-space implementation was done due to time constraints.

7. Reauthentication protocol. The prototype uses DCCP-nonces[DCCP] because it is simple and easy to implement.

8. Simple routing algorithms. Routing subsystem is beyond the scope of the implementation, and therefore a simple static routes-based implementation is chosen.

5 Control and Data Flow

Figure 2(a) data flow at a high level within an entity. Packets enter an entity from one of several possible slots to which the entity listens to. The destination AID contained within each packet is used to select the portal to which the packet must be delivered to. There is one-one correspondence between AIDs, associations and portals. Portal is useful as an abstract notion but there is no data structure associated with it. Packets are directly delivered to the association.

5.1 Control Flow in FARADS Kernel

Figure 4 shows the data flow within the FARADS kernel. Any packet coming in from the network is first decoded to construct an abstract representation of the packet. A routing check is performed to verify if the host is the destination for the packet. If the host is the final destination, the packet is delivered to the slot identified in the destination FD of the packet. If the host is not the final destination, the packet may require special processing before being shipped to the next hop. The packet is then

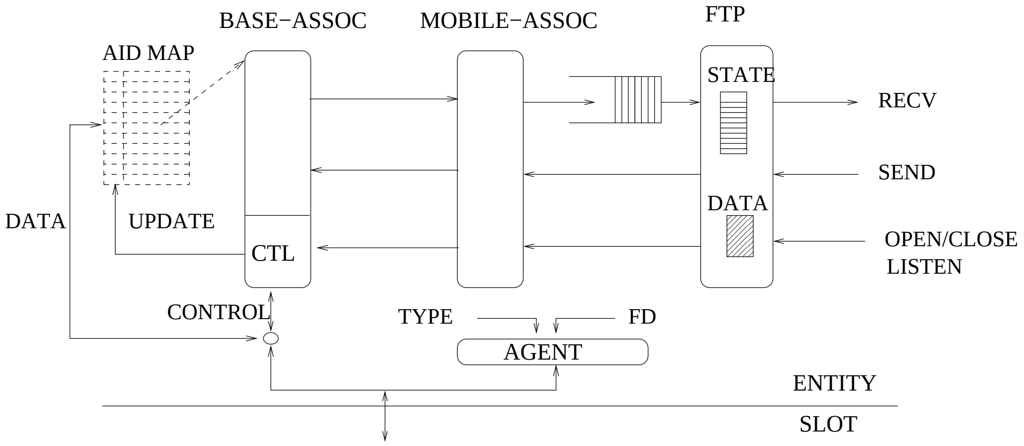


Figure 5: Control and Data flow.

sent to the forwarding function after advancing to the next FD segment in the destination FD. The forwarding function first encodes the packet datastructure before sending the data out. The next hop destination is extracted from the destination FD to select the UDP tunnel that must be used. In our prototype we use IP (version 4 and 6) as the link layer, so an appropriate IP header is constructed and used. The packet processing of outgoing packets is similar to that of the routed packets.

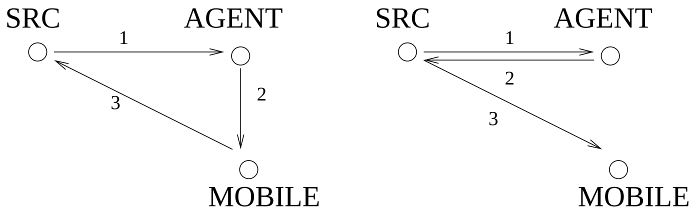
5.2 Control Flow in Entity

Figure 5 shows the control and data flow within entity. The FARADS kernel delivers each packet to the entity as a tuple <Source FD, Destination FD, Data>. FD management is the primary consumer of the source and destination FDs. Entities can probe the FD Management layer for information about the packets such as the number of the slot through which the packet was delivered. The FD management may inform the entity of events such as change in the source or the destination FDs from previous packets.

The entity first checks to see if it is running in the agent mode on the slot of the incoming packet. If so, it looks up an internal data structure for (1) the specific type of agent configured such as redirector or forwarder, and (2) FD of the mobile entity. If the agent is a redirector, a *redirect* message is constructed and sent to the entity at the source FD. In the agent is in forwarding mode, the destination FD is rewritten, and the packet is reinjected back into the network.

If the entity is not running in the agent mode, the destination AID specified in the packet is used to identify the correct association to deliver the packet to. The prototype implementation supports different associations providing different abstractions. Section 8 discusses the protocols used in more detail. A base *connected association* is responsible for handling handshake with the remote end during connection establishment. A *mobile association* can be configured to receive data messages from the base association. The function of the mobile association is to provide transparent mobility through detection of movement, and invoking reauthentication mechanisms upon such a successful detection. In the default case when the data arrives on an expected slot with the correct FDs, the data is delivered to a message queue after stripping the mobile association headers.

The FARADS Transport Protocol (FTP), provides a reliable byte stream abstraction. Internally FTP uses headers which have TCP-like header format. The data extracted from the message queue is a FTP message, and the payload of the FTP message is copied to the user-specified buffer.



(a) Forwarding agent (b) Redirector agent

Figure 6: Agent types

6 Mobility Support

Section 2 introduced the agent-based mobility support. The agent serves as a rendezvous point between a mobile entity and other entities by keeping track of the location of the mobile entity and letting other entities know about it. Existing DNS already serves as a rendezvous point between mobile entities. The mobile entity can, for example, update the DNS whenever it moves to a new location. Commercial implementations such a service already exist. However, it is effective only when the movement occurs infrequently. For high rates of mobility, a third party entity, an agent, is used.

The agent operates as follows. The mobile entity discovers the agent, and registers an FD. The agent responds with another FD which the mobile entity advertises through the DNS. Any other entity which looks up the DNS for the mobile entity's name finds the agent's FD. Any message sent to the agent will be either forwarded or responded to with a redirect message depending on how the agent is configured. The two schemes are discussed below in more detail. The agent has a functionality that is both above and below the line simultaneously. The above-the-line functionality relates to maintaining an association with the mobile entity and and state related to mobile entity's location. The below the line functionality relates to handling of packets meant for the mobile entity. The agent uses the slot number on which a particular packet has been received to distinguish between packets meant for itself and those meant for the mobile entity. Thus an agent listens to atleast two slots. It listens on multiple slots if it serves as an agent for multiple mobile entities.

Two possible kinds of agents are as follows:

FD-Rewriter Agent: This kind of agent simply rewrites the header of packets that are intended for the mobile entity with the mobile-entity's FD and reinjects the packets into the network. Upon receiving such forwarded messages the destination can respond directly instead of the going through the agent. The packet exchange is shown in Figure 6(a). If the connection breaks, the connect can fall back to the default scheme, i.e., route packets through the agent. The disadvantage of such a scheme in which the synack-equivalent message can come from any arbitrary source FD could potentially be a security problem, and therefore will require stronger authentication.

FD-Redirector Agent: When the agent is sent a connection open message or an explicit lookup, the agent responds with a redirect message specifying the mobile entity's FD. The source uses this new FD to send a fresh connect message. The packet exchange is shown in Figure **??**. If mobile-fixed connection breaks, the fixed entity can obtain the latest FD by performing a lookup and executing the reauthentication algorithm. The disadvantage of such a scheme is that anonymous one-way communication, e.g., used in streaming media or multicast, cannot be supported.

When a redirect message is received by the source, the particular send message which resulted in the redirect response must fail or an asynchronous signal must be sent to the application. In either

MyFD()	Compute an initial set of FDs
FDMake(<preferences>)	Create an FD which satisfies the preferences.
FDChar(fd, characteristics)	Evaluate the FD with respect to the identified characterististics.
FDMerge(FD1, FD2)	Combine two FD fragments.
FDSort(<FD set>, <preferences>)	Sort FDs based on the identified characteristics.
FDSplit(fd, <preferences>)	Extract components of the FD based on geographic or topological constraints.
FDOther(fd, <preferences>)	Compute a small set of alternative FDs for a given FD based on some prespecified contraints, e.g., cost.
FDRoutable(fd)	Make the given FD globally routable.
FDDuplicate(fd)	Duplicate the FD
FDAdvance(fd)	Advance to the next segment within the FD
FDTestEnd(fd)	Test whether
FDNextHop(fd)	Compute the next hop

Table 2: FD manipulation functions

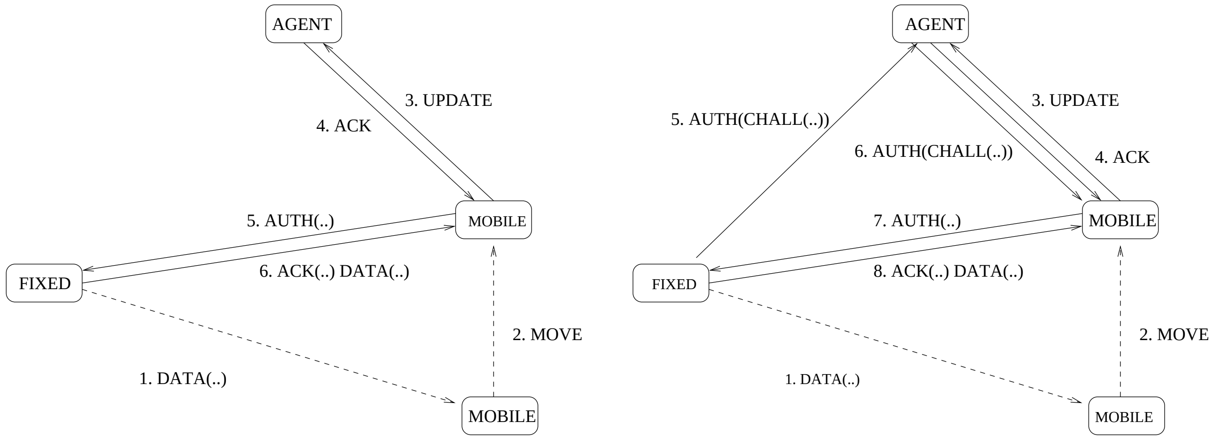
case, the new FD must be provided so that the entity can update its datastructures. The prototype implementation differs slightly from this model. A entity-level redirect control message is generated which is handled by the entity itself.

6.1 FD Management

Describe the FD-Management interface. The assumptions that the entity makes about the FD management functionality such as obtaining an FD, and obtaining FD to a remote location (where the entity is scheduled to move to). Table 2 shows a partial list of functions that must be supported either by the FD management or the routing subsystem. The interface definition is future work.

6.2 Reauthentication

Separation of location and identity in FARADS introduces security issues. FDs are opaque sequence of bits, and the destination to which an FD leads to may not be predictable. Strong cryptographic key-based entity identification will be necessary, but may be computationally too expensive. The prototype assumes a simple, relatively less secure, scheme in which every packet is not authenticated but rather end points can be forced to authenticate themselves at any point during communication,



(a) Initiated by mobile entity.

(b) Initiated by fixed entity.

Figure 7: Reauthentication message sequence.

if necessary. The authentication protocol defines the credentials that must be presented and building up appropriate state at end points to evaluate the credentials. A *challenge* is a request to the remote end point to present the credentials, and a *challenge response* contains the credentials from the remote end point.

Authentication challenge can sent in circumstances such as when the source or destination FD used by an endpoint changes and when a threshold amount of time has passed since that last data exchange. Movement of an endpoint causes a change in atleast one of source or destination FD that is used. Figure 7 considers a scenario where there is one fixed entity, one mobile entity and an agent, and the mobile entity moves. The figure shows the sequence of messages exchanged during reauthentication which is invoked after the movement. Here we assume an unpredictable move by the mobile endpoint, and that the agent is operating is forwarding mode. Figure 7(a) and 7(b) show the exchange when reauthentication is initiated by mobile entity and the fixed entity respectively. On both the sequences, the mobile entity updates the agent first and then presents credentials to the fixed entity. This presentation of credentials may be preceded by a challenge from the fixed entity.

7 Canonical Routes

Support for mobility requires that the routing subsystem or some other subsystem be able to translate between FD's. Suppose FDdst was being used to communicate with a remote entity. When the entity moves to a new location, a new FDdst' must be computed which results in packets being delivered exactly to the same location that FDdst would have delivered the packet to. Now, if the internet consists of a flat network of private domains, then this FD translation could be arbitrarily complex. The complexity increases further if we assume that FDs have QOS-related semantics.

To avoid this complexity, we assume a two level routing domain hierarchy as shown in Figure 8 in which there is a single globally routable domain. End-to-end routes are composed of two FD fragments, viz., FDup and FDdown. FDup corresponds to the part of the FD that results in a packet being delivered in the globally routable domain. The FDdown is the part of the FD that results in a packet being delivered to the appropriate private routing domain. We call the FDdown the *canonical route* because it is invariant between the various end-to-end routes used to reach the destination entity. Given a canonical route to a destination, it is easy to compute an end-to-end FD. The routing subsystem uses local knowledge to compute FDup and concatenation this FDup with FDdown. We can thus avoid most of the FD translation complexity.

We can use canonical routes where ever possible in the existing system. For example, they can be advertised through the DNS or carried as the source FD in messages. Canonical routes are now part of the association state. When an entity moves to a new location, the entity presents the saved canonical route to the destination to the routing subsystem which uses its local knowledge about FDup' at the new location to an compute end-to-end FD.

Efficient routes can be composed if the routing subsystem has additional knowledge about the topology

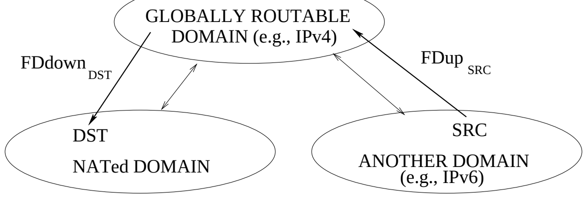


Figure 8: Assuming a single globally routable domain simplifies FD translation

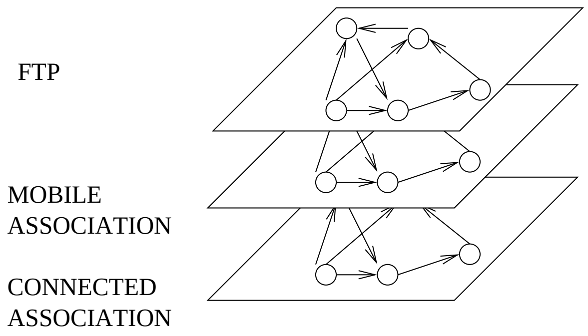


Figure 9: Protocol layers

8 Protocols

The prototype supports four different kinds of associations – simple association, connected association, mobile association and fTP. They are in the increasing order of complexity, shown in the form of protocol layers in Figure 9. A simple association's functionality is similar to that of UDP – unordered, unreliable message sequence. A connected association includes a handshake to establish and destroy the connection. It is also unreliable, but allows for admission control and authentication. Mobile association builds on the connected association to provide transparent mobility by performing authentication and reauthentication upon each move. NewArch implementation of TCP, fTP, adds reliability and ordering. fTP is a simplified version of the standard TCP.

The connection-initiating end must know, in the prototype, the protocol that is being used at the remote end point. This can easily eliminated by introducing a protocol field.

8.1 Simple Association

There are two possible simple associations. First, the class of associations that is equivalent to UDP connections, i.e., unordered datagram delivery service without explicit connection setup. Second, single packet association setup. The prototype support only the first type. Section 9 discusses the packet format in more detail, and the appendex discusses the appropriate field values in more detail.

8.2 Connected Association

Connected association provides unreliable, datagram delivery service but with a single-round trip connection establishment phase. Although the protocol does not provide reliability, a handshake can help identify the source and destination to each other.

Figure 10(a) shows the finite state machine corresponding to the connected association protocol[1]. The protocol state machine at each end point starts in the LISTEN state. Depending on whether that end is initiating the connection, the state machine goes through the WAIT state. Similarly during termination of the connection, the active end (initiator of the termination) goes through the WAIT state.

This protocol provides minimal security. Data is always sent to the destination FD that was used during the connection setup. However, data can be received from any source allowing for certain kinds of security attacks.

[1]The labels have the standard format of "input/output"

STATE	SEMANTICS
OPEN	full fledged connection established
I_WAIT	Wait for the ack from remote end for an explicit move message
I_AGENT_WAIT	wait for the agent's reply
I_REAUTH	reauthentication must be initiated
I_AUTH_SENT	local end's authentication information sent
N_REAUTH	wait for the other end to initiate the reauthentication process

Table 3: Various mobile association states and their semantics

8.3 Mobile Association

Mobile association allows for mobility of the connected association and at the same time provides weak security. Mobile association invokes a reauthentication mechanism whenever one end is unsure about the status of the remote end. This could happen when there is a timeout or which the source of a datagram has changed We the DCCP nonces as the authentication mechanism, and use the rendezvous strings to exchange the nonces.

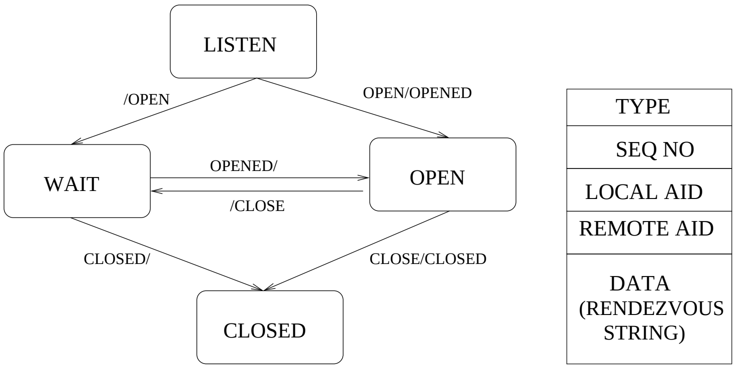
State transitions in figure 11(a). The state machine starts in the OPEN state when the underlying connected association completes the handshake, and enters the OPEN state. Mobile association state transitions occur while the connected association remains in the OPEN state. When a close is invoked on the mobile association, the necessary cleanup is performed and a close operation on the connected association is invoked.

In Figure 11(a), states are prefixed by either I or N to indicate whether they are the initiator or non-initiator ends of the association. The following transitions occur at the initiating side:

8.4 fTP

The FARADS Transport Protocol (fTP) provides for reliable byte stream. We are interested only in the reliability aspects and not the performance. As such we use window sizes of 1 byte and fixed timeout intervals for retransmission. Figure 12(a) shows the state machine that was implemented. The packet loss testing tool Sting [Sav99] was modified for the purpose.

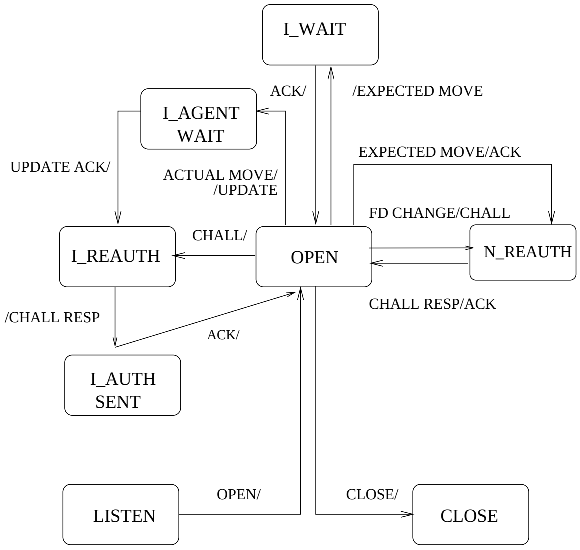
fTP header is identical in syntax and semantics to the standard TCP header. However, we use a restricted range of possible values to the various fields and parameters. In particular we ignore the congestion control algorithms.



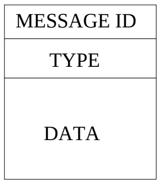
(a) Connected association state machine

(b) Header format

Figure 10: Connected association components



(a) Mobile association state machine FIX ME



(b) Mobile header format

Figure 11: Mobile association components

The checksum of the fTP packet is computed over the pseudo-header consisting of the local and remote AIDs. AIDs are equivalent to the source and destination port numbers in TCP in IP networks. This provides some, albeit small, improvement in security by preventing another association on the remote host from sending data.

Data is delivered to the user, i.e., a *recv* call will return when TCP sees a PUSH flag set.

By advertising a window of 1 byte, the prototype, although inefficient, can ensure that all data is received with out complicated reordering-related problems.

8.5 Discussion

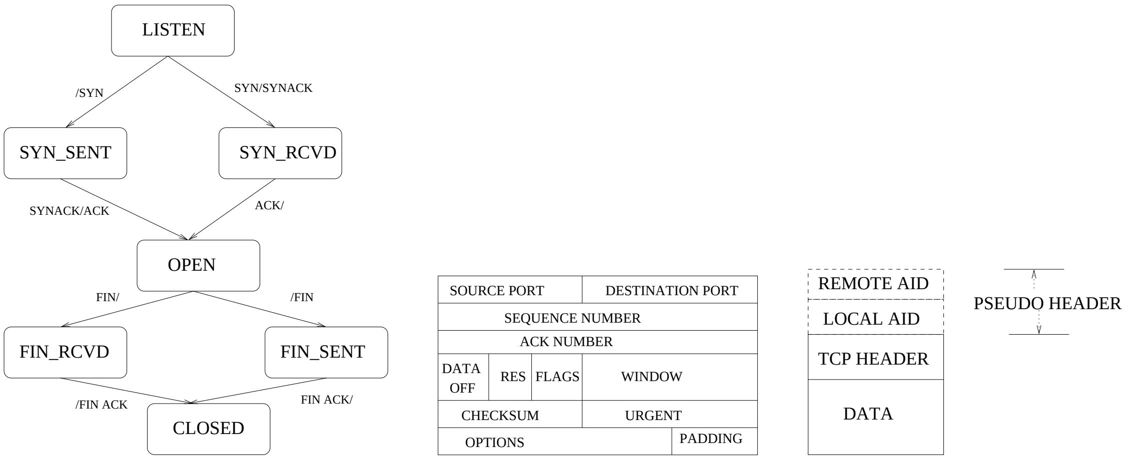
The inter-layer dependencies results in additional programming complexity. They are as follows:

1. Connection open and close: Each layer has a associated handshake to be performed. However we cannot serialize these handshakes because such a design will require as many round trips as there are layers. Therefore the handshake messages carry tokens from all the different layers.

2. Dependencies: The pseudo-header computation at the fTP layer requires the AIDs to be known in advance. However, the aid is assigned by the connected association layer. This requires separation of the AID selection from the association establishment process.

9 Packet Formats

Figure 13(a) shows the default on-wire format of packets in FARADs. The destination FD identifies the entity to which the packet must be delivered to, and the source FD identifies the return path to the entity sending this packet. Each FD has two components – a routing directive and a slot number. The routing directive helps the packet traverse the network to reach the destination physical host,



(a) fTP state machine

(b) TCP Header

(c) Mobile packet's fTP payload

Figure 12: Header formats

and the slot is use to identify the correct entity at that destination host. Figure 14 presents a small subset of possible forwarding directives.

The semantics of the rest of the fields are as follows: The link layer header is a per-hop header. Since our implementation is an overlay network, we use IP as the link layer. The source and destination AIDs are used to select the correct association within the receiving entity to handle the packet. The TTL is used to control the maximum number of hops that a packet can traverse, and is decremented after each hop.

The default packet format is similar to IP, and provides no guarantees. Mobility and reliability are provided by appropriate protocol layers, mobile association and NewArch TCP respectively, within the association. Headers corresponding to them are shown in Figure 13(b) and 13(c). Note that TCP header and the mobile header are independent, but the figure shows the typical use. The details of the headers are discussed in Section 8.

9.1 Header Formats

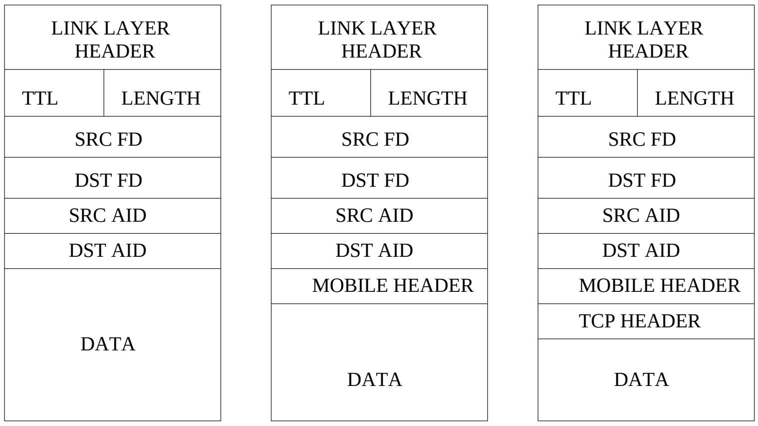
Section 8 discussed the various header in more detail in the context of individual protocols. So we do not discuss them here.

9.2 FD Formats

FDs are constructed and manipulated by the routing subsystem and the FD management layer. The entity treats the FD as an opaque sequence of bit but the structure is visible to the routing and forwarding functions below the line at end hosts and the routers. After bootstrapping, an entity queries the routing system for a globally routable FD to itself and registers the returned FD with the NewArch DNS. A client wishing to find the entity performs a DNS lookup.

A linear structure in which the there are series of FD components is assumed by the FARADs kernel.

The prototype has been tested with a variety of FD formats. The length of the FD depends on the FD type, and we expect the FARADs network to support a small number of FD formats. Figure 14 shows the different FD formats supported by the prototype. IPv4 FD and IPv6 FD have a simple

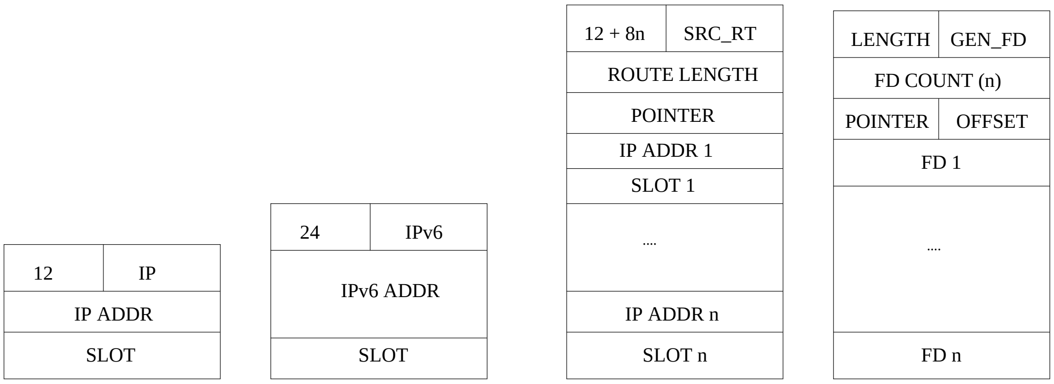


(a) Default packet format

(b) Mobile association packet format

(c) fTP packet format

Figure 13: Various packet formats



(a) IPv4 forwarding directive

(b) IPv6 forwarding directive

(c) Source route FD

(d) Generalized FD

Figure 14: Various forwarding directives

format with fixed length. The IPv4 source route FD, on the other hand, has variable length. Note that this is not source route in the traditional sense because by specifying non-default slot numbers for intermediate nodes, the source can allow for processing at intermediate nodes. The generalized FD shown in Figure 14(d) consists of a series of FDs, each of which can be an IP FD or source route FD. It has a linear structure consisting of network segments that must be traversed in the specified order.

10 NewArch DNS

NewArch DNS in the FARADS prototype is an entity. The FARADS kernel is assumed to be configured with the DNS FD. FARADS DNS supports a simple mapping between a name and a canonical FD described in Section 7. The name is free form and only exact matching is supported. Section A.8 presents the DNS registration and lookup protocol in more detail.

11 Experimental Scenarios

This section presents three scenarios – route mobility, host-level mobility and agent placement to show the flexibility of FARADS. All of them have been implemented and tested in the prototype.

Route Mobility

Route mobility, also called virtual mobility, involves switching between FDs without destroying the already established association. A sample sequence of operations for an experimental setup shown in Figure 15 include:

1. FIXED establishes connection with MOBILE (at 10.1)
2. MOBILE instructs FIXED to use a different FD (20.1)
3. FIXED initiates process of reauthentication with MOBILE
4. FIXED and MOBILE resume data transfer

Host-Level Mobility

Host-level mobility is the traditional notion of mobility where the mobile host, potentially containing more than one entity, physically moves to a different location. If the movement is predictable, then the mobile entity can suspend the association until the move is completed and resume it after reauthentication at the new location. If the move is unpredictable reauthentication process can be initiated by either the non-mobile or the mobile entity. In case of the former, the non-mobile entity first contacts the mobile entity's agent to obtain the current location of the mobile entity before initiating the reauthentication process. A sample sequence of operations for an experimental setup shown in Figure 15 include:

1. FIXED establishes connection with MOBILE (at 10.1)
2. MOBILE's host's 10.1 interface goes down and therefore MOBILE is unreachable.
3. FIXED goes back to contact the agent to enquire about MOBILE's new FD
4. AGENT returns MOBILE's FD corresponding to 20.1
5. FIXED contacts MOBILE at the new FD.

12 Future Work

The exiting prototype supports different forms of FDs, agents, and associations. It can be extended in several different ways. They include:

1. The prototype implementation supports only one fTP instance per entity. The prototype must be extended to support multiple fTP instances.
2. Remove the constraint of one FARADS host per physical host

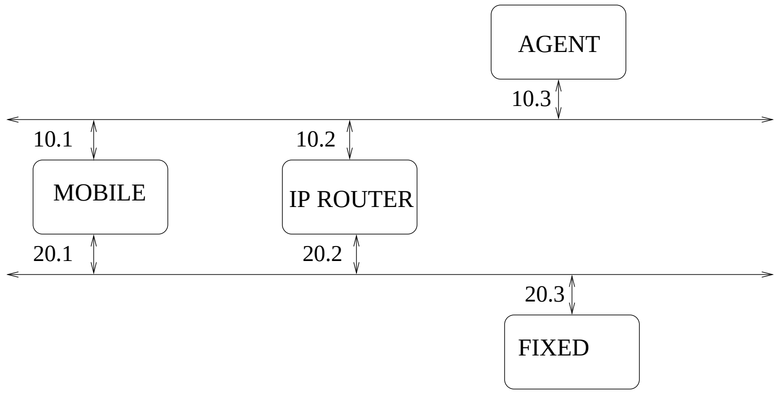
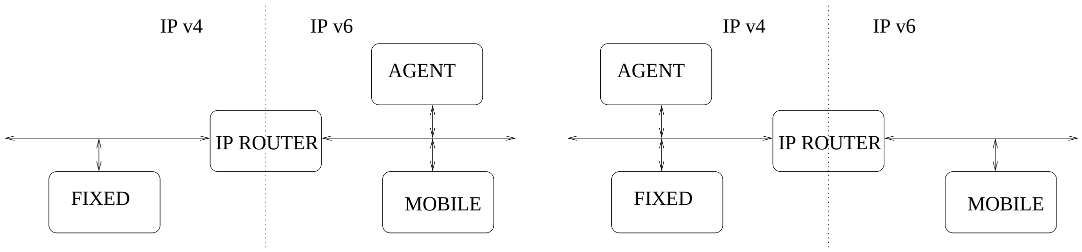


Figure 15: Basic experimental setup



(a) Scenario 1

(b) Scenario 2

Figure 16: Other experimental scenarios

3. Allow the agent forwarding to happen simultaneously with the
4. Single packet association setup.
5. Extend FARADs kernel to be multihomed beyond one V4 and V6 interface.
6. Support multicast.
7. Implement tools to add/delete/modify routing table entries.

References

[Cla02] David D. Clark. Forwarding Directives, Associations, Rendezvous and Directory Service (FARADS). April 2002.

[FBP02] Aaron Falk, Robert Braden, and Venkata Pingali. A FARADS-based Network Design, September 2002.

[KHF03] Eddie Kohler, Mark Handley, and Sally Floyd. Datagram Congestion Control Protocol (DCCP). Work in Progress. Internet Draft, March 2003.

[Sav99] S. Savage. Sting: a TCP-based Network Measurement Tool. In *Proceedings of USENIX Symposium on Internet Technologies and Systems*, October 1999.

APPENDIX

A Programmers Manual

This document discusses the FARADS prototype implementation in more detail. It presents the interfaces to various abstract data structures used in this prototype, and discusses important implementation details. As such, this document assumes familiarity with the FARADS [Cla02, FBP02] terminology and and the high level design of this prototype.

The appendix first presents an overview of this prototype, and follows it up with a detailed discussion of the implementation of the various pieces of the architecture. The outline of the rest of the appendix is as follows. Section A.10 presents a simple example FARADS application. Section A.1 introduces the source files and compilation framework. Section A.2 discusses the implementation of slots. Section A.3 presents the interface to forwarding directives. Section A.4 discusses the various types of associations supported and their respective interfaces. Section A.5 discusses the various mobility-related messages that are exchanged between entities. Section A.6 presents the simple routing system that this prototype supports. Section A.7 presents the interface to the authentication module and discusses a specific implementation. Section A.8 discusses the simple naming system that this prototype supports. Section A.9 discusses some helper applications. Section A.11 summarizes the implementation and identifies possible future work.

A.1 Source Overview and Build

The source distribution consists of the C++-language source files, and scripts. Executing *configure* script results in a *Makefile* which can be used with UNIX *make* utility.

The source files can be broadly classified as belonging to the library or to an application. Table 4 lists the various source files that generate the library *libfarads.a*, and identifies their contents. Table 5 presents the various application source files, and the various executables produced corresponding to these application source files. The source distribution contains instructions for using these executables.

File	Contents
slot.cc, slot.h	Implementation of slots
association.cc, association.h	Implementation of association base class *association_base* and *connected_association*
aa_association.cc, aa_association.h aa_request.h	Agent aware connected association, *aa_association*.
mobile_association.cc, mobile_association.h mobile_queue.h	Implementation of the mobile association
fTP.cc fTP.h	Implementation of FARADS transport protocol
mobile_message.cc, mobile_message.h	Implements the message formats
inet.cc, inet.h, checksum.h	Provide helper functions to TCP-format packets
farads.h	Global include file
authentication.h	Defines the interface to authentication mechanisms and a specific authentication mechanism implementation, DCCP.
serializable.h	Defines an interface that can be used to convert and object to and from network independent representation.
counted.h	Provides reference counts to derived classes.
fd.h	Implementation of the various forwarding directive types

Table 4: libfarads.a source files

Figure 17 shows the relationship between the various important classes. In the rest of this document we discuss the various classes in more detail.

A.2 Slot

Source Files: slot.cc, slot.h

Slots are the FARADS kernel multiplexing/demultiplexing points. Incoming messages are received on a slot, and outgoing messages are sent on one. The slot is an abstract data structure built on top of a local UNIX socket. A reference count is used to keep track of the number of references to the data structure. A slot is closed and deallocated when the the last reference goes out of scope or is deleted. Slots can also be closed explicitly.

Once a slot is allocated, all the operations in Table 6 are supported. Most applications will not use slot operations directly, however. Most associations should know what slot they operate on and hide this from the user.

Each slot is identified by a 32-bit integer, and there is a constructor that takes such an integer, used to create or access allocated slots. The 32-bit integer decision is arbitrary, and should be considered an artifact of implementation. Constructing a slot object with a given integer adds a reference to the underlying socket if the slot already is allocated, or exchanges messages with the local *faradd* to allocate the slot if not. If a slot is required, but the application can use any slot number, passing a special integer, *slot::NO_SLOT*, to the constructor tells the *faradd* to select the slot number. A slot

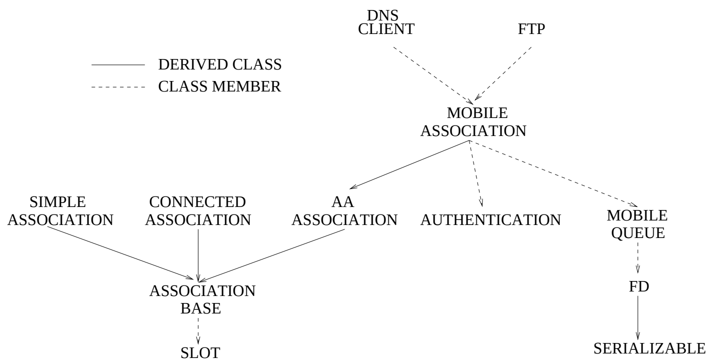


Figure 17: Relationship between some important classes

File	Executable	Contents
dnsserver.cc dnsserver.h	dnsserver	FARADS DNS service
dnsclient.h	-	Library interface to call the dns
farads_exceptions.h	- -	implementation of the exceptions throw in case thrown
farads_debug.h	-	Implementation of the debug class to control the level of reporting.
routing.h	all	Implementation of the routing subsystem
faradd.cc	faradd	FARADS kernel
packet.cc packet.h	faradd	Implementation of the abstract representation of the messages within the *faradd*
aa_ping.cc aa_pong.cc aa_agent.cc	aa_ping aa_pong aa_agent	Simple ping application used to test a mobile association. Uses command line options and a single domain. aa_agent is an application implementing the agent functionality.
agent.cc fixed.cc mobile.cc	agent fixed mobile	Echo client and server with agent support. Uses *mobile_association* and simulates host-level mobility.
fixed_fTP.cc mobile_fTP.cc	fixed_fTP mobile_fTP	Echo client and server. Uses FARADS transport protocol, *fTP* instead of the *mobile_association*.
ping.cc pong.cc	ping pong	Simple echo server and client. Uses connected association
testdnsv2.cc	testdnsv2	DNS service test application.
uping.cc upong.cc	uping upong	Simple echo client and server. Uses *simple_association*

Table 5: Contents of the important application source files

Prototype	Effect
uint32_t number()	Return the slot number
void send(const void *b, int len, const forwarding_directive& fd) throw(slot_exception)	Send the buffer to the given FD. Throws an exception on an error.
void recv(const void *b, int len, const forwarding_directive& fd) throw(slot_exception)	Receive a buffer. *fd*is set to a reply FD. This is a blocking call. Throws an exception on an error
bool message_waiting()	Return true if a *recv* call will return immediately.
void close()	Explicitly close the slot. No more packets will be received, nor can any be sent on the slot after this call is made.

Table 6: *slot* operations

Name	Semantics	Value
IP	IP v4 FD	1
IPv6	IP v6 FD	2
SRC_ROUTE	IP v4 source route	3
GEN	Generalized FD	4
NO_TYPE	Null FD	9999

Table 7: FD types

object can be created without allocating a slot number from the *faradd* using the default constructor. Slots can be assigned to and initialized from other slots.

The protocol for allocating a slot is straightforward. Messages travel between the entity and the *faradd* via a local, i.e., UNIX-domain socket. Each message contains a message of the format in the *slot::request* class. To allocate a slot, an *ALLOCATE*, message is sent. If the allocation is successful an *ALLOCATED* message is returned or an error is indicated by *NOT_ALLOCATED* message. Analogous messages exist for deallocation. All messages are sent on the same channel to avoid ordering problems. Once either request is made, the entity waits 0.5 seconds for a reply. If none comes, the request is assumed to have failed. The protocol is not very robust to errors, but the combination of high reliability of local sockets and simplicity of implementation argues for a simple protocol.

A.3 Forwarding Directives

Source Files: fd.h

Forwarding directives (FDs), as described in the companion document, are set of instructions that can take a packet from a source to a destination. Forwarding directives have two parts. First, a network part that helps the packet traverse the network, and second, an end system part which is used to select the correct entity within an end system. In this prototype, the network part of the FD consists of an ordered sequence of IPv4 and IPv6 addresses. The end system part in our implementation is a 32-bit slot identifier. This prototype supports different kinds of FDs, but in all cases the end system part, slot identifier, is fixed. The on-wire formats of the various FD types described in Section 9.

FDs are created by either the FD management layer or the routing subsystem. Section A.6 discusses the routing system interface provided for this purpose. Entities treat FDs are an opaque sequence of bits. In this prototype most applications use the routing system to obtain FDs, and deal with only references to FDs. Operations such as duplication, comparison, and sorting are ideally done by the FD management layer, but the abstract data structure used to represent FDs supports these operations directly within the entity.

This prototype supports five types of forwarding directives: Null, IPv4, IPv6, IPv4 Source Route, and Generalized. Table 7 specifies the types and values corresponding to the various FD types. Within the entity and the FARADS kernel, the FD is represented as an abstract data structure, *forwarding_directive*. Table 8 specifies the common interface to all FDs. The various FD types have different constructors. The IPv4 and IPv6 FD constructor take as input IPv4 and IPv6 addresses respectively. The IPv4 Source Route FD constructor requires the specification of the source and destination IPv4 address. The Source Route FD constructor invokes a helper function to compute the source route. A Generalized FD can be constructed by specifying an ordered list of FDs.

The forwarding directive interface methods can be broadly classified as being delivery-related, format conversion-related, and user-interface related.

The routing subsystem uses the delivery-related FD functions. These include the *in_addr* method and the *advance* method. The *in_addr* function returns an *address* structure containing the next hop in a specific domain. Currently the *address* structure supports only IPv4 and IPv6 nexthops. It is executed by each FARADS router along the path from the source to the destination. When

bool my_type(void *, int) const	Check if encoded FD has my type
uint32_t my_type() const	Return my type
bool operator==(const forwarding_directive& f) const	Compare two FDs for equivalence
forwarding_directive *make_copy()	Create a duplicate
address *in_addr() const	Return next hop address (e.g., an IPv4 address)
uint32_t slot() const	Return the slot identifier
void advance()	Forward internal pointers to next FD segment
void post_processing()	Any post processing (e.g, resetting pointers) of FD to be done before making storing it

Table 8: Important *forwarding_directive* methods

the packet reaches the next hop identified by the address structure returned by *in_addr* method, the *advance* method is invoked to set an internal pointer within the FD to point to the next FD segment. A packet is assumed to have reached its destination if even after the advance operation, the local end system/router still remains the destination. The method *slot* returns the slot number associated with an FD. This is used by the farads kernel to identify the entity to which the data must be delivered, and the OS data structure corresponding to it.

The representation of an FD within the packet is different from the representation within an end system. The *forwarding_directive* class provides methods *serialize* and *unserialize* to perform conversion between these different forms. A helper method *my_type* extracts the type of the FD when it is the decoded, i.e., when the FD is in normal form, and tests whether the type of the encoded FD is the same as another FD. This is useful in determining the correct domain for taking the next hop and the appropriate gateway. An artifact of the implementation is that an end system cannot simultaneously be resident in multiple V4 or V6 domains at the same time. This problem can be eliminated by providing some context for interpreting the V4 address which can disambiguate between different structurally similar routing domains.

The entity stores source FDs of incoming packets to allow for sending responses. The *make_copy* function, as the name indicates, is used to make copies of the forwarding directives for storage and manipulation. Before storing FDs, any internal pointers may have to be reset. The *post_processing* invoked just before storing an FD allows for resetting of such pointers.

A.4 Associations

This prototype supports various types of extensible associations, and each association is implemented as a C++ class. The relationship between the various kinds of classes is shown in Figure 17. In this section we discuss each of the associations in more detail.

A.4.1 Class *association_base*

Source Files: association.h, association.cc

The class *association_base* is the base class identifying the minimal interface that other derived associations must support. The interface is very simple, and is shown in Table 9. The *send* and *recv* methods invoke *send_message* and *recv_message* respectively that are implemented by derived classes such as *mobile_association*. The method *rv_string* returns the rendezvous string sent by the remote end during connection establishment.

A.4.2 Class *simple_association*

Source Files: association.h, association.cc

Prototype	Effect
string rv_string()	Return the rendezvous string
void send(const void *b, int len) throw(farads_exception)	Send the buffer on this association. Throws an exception on an error.
void recv(const void *b, int len) throw(slot_exception)	Receive a buffer on this association. This is a blocking call that throws an exception on an error.

Table 9: *association_base* operations

Prototype	Effect
void send(const void *b, int len, const forwarding_directive &fd) throw(farads_exception)	Send the buffer on this association to the specified destination. Throws an exception on an error.
void recv(const void *b, int len, forwarding_directive &fd) throw(slot_exception)	Receive a buffer on this association and return the source FD using the specified fd. This is a blocking call.Throws an exception on an error.

Table 10: Methods added by *simple_association*

The *simple_association* provides connectionless transport similar to UDP. The *simple_association* class adds *send* methods that includes a destination FD, and a *recv* method that can return the source FD. Other than that, the basic association model is preserved. The *uping* and *upong* applications use *simple_associations*. This class of associations do not use association IDs (AIDs).

A.4.3 Class *connected_association*

Source Files: association.h, association.cc

connected_association provides a connection oriented transport without reliability or ordering guarantees. It defines a simple two-messages protocol between two *connected_association* end points to open and close an association presented in Section 8.2. Table 11 shows the various methods support by this class. *connected_association* defines a constructor that just takes a slot and rendezvous string that results in a passive association end point. Calling *listen* on that association returns a fully formed connection should one become available. The active open is accomplished by using a constructor that takes a slot, rendezvous, and the FD of the other end. This constructor initiates the connection establishment protocol. The *listen* and active constructor each allocate an association identifier on either end and set up appropriate demultiplexing state to provide separate message streams over the same slot. The *ping* and *pong* applications use *connected_association*.

The method *process_message* receives all incoming messages and classifies them into control, valid data and invalid data messages based on the destination AID. Control messages are handed off to *control_message* method. In case of valid data messages, the headers are stripped, and the body enqueued in a message queue. Invalid data messages are dropped.

A.4.4 Class *aa_association*

Source Files: aa_association.h, aa_association.cc

aa_association is similar to *connected_association* in that it provides connection oriented, unreliable datagram service.

aa_association class is constructed out of *connected_association* code base but is not a sub class. *aa_association* has two major changes from *connected_association*. It handles the mobility-related redirect message, and supports a variety of FDs by converting all references to pointers. This class is intended to be used with *mobile_association*. *connected_association* can theoretically replace by *connected_association* which has been retained for historical reasons.

Table 12 shows the various methods support by this class. The semantics of the *process_message* method is almost identical to that of *connected_association* with one difference. The agent aware

Prototype	Semantics
connected_association listen()	Accept incoming connection and return a newly instantiated association
int send_message(const void *, int) int send_message(const void *b, int len, const forwarding_directive&)	Send data through the slot
int recv_message(void *, int)	Receive data from the slot
connected_association listen()	Accept incoming connections and return a a newly instantiated association.
void process_message()	Process incoming packets

Table 11: Important *connected_association members*

association can also be set to function in an agent-mode. In this mode, all incoming messages from specific slots on which the entity is listening are handled as an agent is supposed to - rewrite the header and reinject the packet into the network. This agent function is embedded in the the *process_message* method. A common data structure is maintained to store agent-related information such as the slot, type of agent and mobile entity's FD. *aa_association* provides a low level interface to manipulate this common data structure. The interface consists of the *add_forwarding_rule* and *delete_forwarding_rule* functions.

A.4.5 Class *mobile_association*

Source Files: mobile_association.cc, mobile_association.h, mobile_queue.h

The *mobile_association* class is derived from *aa_association* class, and adds support for mobility. Table 13 shows the various methods support by this class. A *mobile_association* can be instantiated in a way similar to *connected_association* and *aa_association*. The only difference in the constructor interface is that a *mobile_association* constructor requires a source FD to be explicitly specified. Constructors that do not require a source FD to be specific such as *connected_association* constructor compute the source FD in the background assuming IPv4 routing domain and using the first interface's address. The major changes from the base class *aa_association* include introduction of a new mobility header, extension of the stateful message queue with association-related state, additional functions for mobility protocol processing, and extended user interface.

As discussed in Section 8, the *mobile_association* adds a new header to the body of the FARADS packets, and uses messages contained in this header to ensure continuous connectivity to a mobile entity as discussed in the FARADS design document [Cla02, FBP02]. The header is used to carry a variety of messages. The message types and semantics are listed in Table 15. Section A.5 discusses these messages in more detail.

Both *connected_association* and *aa_association* deliver all non-control (open/close) messages to a stateful message queue. These messages remain in the queue until they are read by the user. A design

Prototype	Semantics
int send_message(const void *, int) int send_message(const void *b, int len, const forwarding_directive&)	Send data through the slot
int recv_message(void *, int)	Receive data from the slot
aa_association listen()	Accept incoming connections and return a a newly instantiated association.
void add_forwarding_rule(uint32_t slotnum, aa_association_class_t aa_class, forwarding_directive *redir)	Add a mapping between a slot and destination FD (corresponding to a mobile entity). Used when the entity is in agent mode.
void delete_forwarding_rule(uint32_t slotnum)	Delete an existing mapping

Table 12: Important agent aware association (*aa_association*) members

Prototype	Semantics
int send(const void *b, int len_in) int recv(void *b, int len_in)	Basic send/receive functions.
int send_message(const void *, int) int send_message(mobile_queue *q, const void *b, int len) int send_message(mobile_queue *q, const void *b, int len, forwarding_directive &tfd, forwarding_directive& sfd)	Send data on slot
void send_request(const request& r)	Encode the request and invoke send_message
void lookup_agent()	Reset the destination FD to that of the agent and send authentication challenge
void process_message()	Process data coming into the entity
void process_protocol_message(aid_t sa, aid_t da, char *pkt, int pktlen, forwarding_directive *tfd, forwarding_directive *sfd)	Process mobility-related message
bool message_waiting()	Check if there is data on message queue
forwarding_directive* get_fd() void set_fd(forwarding_directive* fd_in) forwarding_directive* get_source_fd() void set_source_fd(forwarding_directive* fd_in) forwarding_directive* get_agent_fd() void set_agent_fd(forwarding_directive* fd_in)	Set and get various FDs. (Wrappers around mobile_queue calls)
uint32_t get_local_aid () uint32_t get_remote_aid ()	Get AIDs. (Used by FARADS transport protocol)
void start_agent(int x)	Execute in agent mode
void reauthenticate()	Send authentication information to the remote end
void move(forwarding_directive *tfd)	Inform the remote end of new FD
ma_set_t association_select(ma_set_t ma_set, struct timeval &to)	Posix select over associations

Table 13: Important methods in *mobile_association*

Prototype	Semantics
mobile_queue(association_state_t st_in, forwarding_directive* source_fd, forwarding_directive* dest_fd)	Create a queue and initialize it with the association state, source and destination FDs
void set_local_aid(aid_t aid) void get_local_aid(aid_t aid)	Set and get local aid. Set and get remote aid.
void set_remote_aid(aid_t aid) void get_remote_aid(aid_t aid)	
void set_fd(forwarding_directive* fd) forwarding_directive* get_fd()	Set and get destination FD
void set_source_fd(forwarding_directive* fd) forwarding_directive* get_source_fd()	Set and get source FD
void set_agent_fd(forwarding_directive* fd) forwarding_directive* get_agent_fd()	Set and get agent FD

Table 14: mobile_queue class methods

decision made early during this prototype implementation was to separate the queue and the rest of the association data structure to allow flexibility in the implementation of the association. In this design, an association ID (AID) maps to this message queue instead of the association object. Support for mobility requires authentication-related and mobility-related messages to be exchanged even when the user is not actively reading from the stateful queue. To allow access to mobile association state at all times, the stateful message queue is extended with necessary association-related state such as the FDs and AIDs, and this new data structure is called the *mobile_queue*. The interface to *mobile_queue* is presented in Table 14. In addition to the stateful message queue operations, the interface provides a series of methods to update the association state. Where necessary, the *mobile_association* methods are modified to use the *mobile_queue*. The implementation of the *mobile_association* assumes a *mobile_queue*.

All incoming messages on a slot are received by *process_message*. Control messages are handed over to the *control_message* method of *aa_association*, and all other messages are directed to the *process_protocol_message*. Control messages are identified using a well-defined destination AID. Non-control messages include data and mobility-related messages. In case of mobility messages such as reauthentication, *process_protocol_message* performs appropriate *mobile_association* state transitions, and responses generated. In case of data, the mobility header is stripped to extract the content, and added to the outstanding message queue.

The association protocol state is stored in the variable *astate*. The method *process_protocol_message* manipulates *astate*. The various states of the association are: OPEN, CLOSED, LISTENER, I_REAUTH, I_AUTH_SENT, I_WAIT, I_AUTH_SENT, N_REAUTH, and N_AUTH_SENT. The description of the states and transitions are described in Section 8

mobile_association also supports a few other mobility-related functions. The method *reauthenticate* is invoked when a mobile entity wishes to update the remote end of an association with a new FD. This might be necessary when the entity moves or when the entity has a preferred route. The *mobile_association* class also provides a *start_agent* function that handles all incoming packet on a specified slot as an agent. The class *aa_association* also provides *add_forwarding_rule* that can be used to specify a slot number, type of agent functionality and mobile entity's FD.

The method *association_select* is equivalent to the unix select system call for associations. This allows for each entity to use create and use multiple associations simultaneously. It takes as input a set of mobile associations and a timeout and returns with the set of mobile association that have data.

A.4.6 Class *fTP*

Source Files: fTP.h, fTP.cc, inet.cc, inet.h, checksum.h

FARADS transport protocol, *fTP*, builds on *mobile_association* by adding reliability, ordering and a byte stream abstraction. The interface to *fTP* is identical to *mobile_association*.

fTP adds a header, that is identical to TCP header, and uses the sequence numbers contained within to order data, detect packet losses. Miscellaneous functions such as *WriteTcpPacket* and *ReadTcpPacket* help manipulate the header. The checksum in the *fTP* header is computed over source and destination AIDs instead of the standard IP pseudo-header. The retransmission algorithm is simple, and based on fixed, pre-defined timeout. Initial sequence numbers are exchanged during association establishment using rendezvous strings.

The current implementation of *fTP* is written as an application on top of *mobile_association*. An artifact of the implementation is that immediate responses to *fTP* messages such as acknowledgments and retransmissions cannot be generated when the entity performs a blocking read or write operation on another association. This is not a problem if all associations are *mobile_associations*. A closer integration of the *fTP* with *mobile_association* is part of the future work.

A.5 Mobile Messages

Source Files: mobile_message.cc, mobile_message.h

Table 15 lists the various mobility-related messages along with their contents. The header is simple with an identifier, type and body. The identifier is used to detect losses and match requests and responses. The type field is self-evident.

The agent register message is sent by a mobile entity to the agent to register its presence. Future extensions to this message type will allow the mobile entity to select from among different agent modes possible. Currently there are two modes. An explicit move message is sent by the mobile entity to a remote end of an association when the movement is predictable and destination of the movement is known. In response to the move message, the remote end prevents any further data transfer until the mobile end reauthenticates itself *after* the move. Reauthentication between two entities is performed

Type	Value	Contents	Semantics
MOB_AGENT_REGISTER	1	id, type, fd	Agent register message
MOB_NEG_FRAGMENT	2	-	Place holder for FD negotiation messages
MOB_DATA	3	id, type, len, data	Data
MOB_DATA_ACK	4	id, type	Data acknowledgement
MOB_REAUTH_ACK	5	id, type	Move acknowledgement
MOB_REAUTH_MOVE	6	id, type, fd, cred	Explicit move message
MOB_REAUTH_AUTH_CHALL	8	id, type, challenge	Authentication challenge message
MOB_REAUTH_AUTH_CHALL _RESP	9	id, type, resp	Authentication challenge response
MOB_REAUTH_AUTH_ACK	10	id, type	Acknowledge of the challenge response

Table 15: Mobile message types

Function	Semantics
forwarding_directive *get_fixed_to_agent_route() forwarding_directive *get_agent_to_fixed_route() forwarding_directive *get_agent_to_mobile_route() forwarding_directive *get_mobile_to_agent_route() forwarding_directive *get_fixed_to_mobile_route() forwarding_directive *get_mobile_to_fixed_route() forwarding_directive *get_fixed_to_dns_route() forwarding_directive *get_dns_to_fixed_route() forwarding_directive *get_mobile_to_dns_route() forwarding_directive *get_dns_to_mobile_route()	Optional functions to compute end to end routes between individual entities
forwarding_directive *get_canonical_mobile_route() forwarding_directive *get_canonical_dns_route() forwarding_directive * get_canonical_agent_for_mobile_route() forwarding_directive *get_canonical_agent_route() forwarding_directive *get_canonical_fixed_route()	Compute canonical routes to individual entities
forwarding_directive *compute_full_route()	Given a canonical route, compute the the end to end route.
void set_location(where_t w)	Tell the routing system where the entity is

Table 16: Routing subsystem interface

using a challenge response protocol. The contents of the challenge, response and acknowledge messages are dependent on the specific authentication protocol.

A.6 Routing

Source Files: routing.h

This prototype supports a primitive form of routing subsystem servicing a few, well defined entities. We assume that the routing subsystem has knowledge about (1) the (only) routing domain hierarchy which is shown in Figure 18, and (2) a small number of network topologies which are shown in Figure 19. The routing system is static both in terms of knowledge and computation. Only the minimum required routing functionality was implemented, and routing system interface and design is a major area of future work.

The routing subsystem presents a simple library call interface shown in Table 16. The interface provides a set of calls to compute the canonical FDs to individual entities. Canonical FDs/routes for a node are the set of instructions that deliver a packet to that node starting in a global routable domain (IPv4, in this case). The interface has a separate call corresponding to each entity because the routing subsystem does not have any way to identify the entity invoking the routing subsystem. The *compute_full_route* method translates these canonical routes into complete end-to-end routes. However, the routing subsystem requires knowledge about the location of the entity. The location can explicitly specified using the *set_location* call.

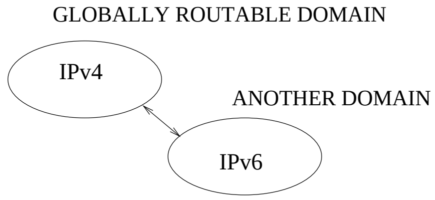
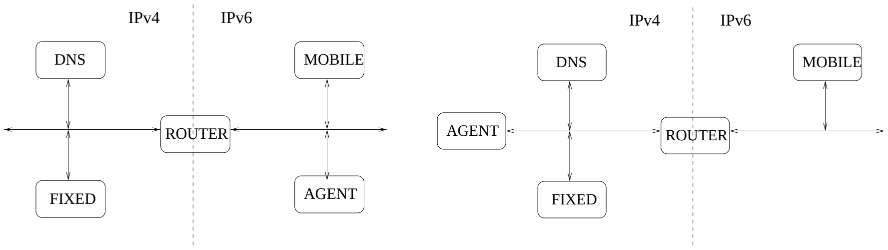


Figure 18: Routing domain hierarchy supported by this prototype.



(a) Agent located in the IPv6 domain

(b) Agent located in the IPv4 domain

Figure 19: The static topologies supported by the routing subsystem

A.7 Authentication

Source Files: authentication.h

Separation of location and identity requires authentication to offset at least some of the security problems introduced by the separation of the location and identify. A connection end point can reauthenticate itself or require reauthentication of the remote end point anytime during communication. An end point generates a authentication challenge requesting the remote end point to prove its credentials, to which the remote end point responds with an challenge response. In both cases the messages are tested for validity. An end point may unilaterally send a challenge response usually to update destination FD information. This prototype defines an interface and contains an implementation of DCCP [KHF03].

Table 17 lists the various methods that are part of the interface. The methods *init_send* and *init_recv* are used to initialize the authentication state at either end. The method *init_send* is used to generate an authentication string to be sent to the remote end, and *init_recv* is used to update the local module's state with authentication data from the remote end. The authentication data exchange uses the rendezvous string[2], but *mobile_association* treats the data as an opaque character string. The

[2]The rendezvous string has structure imposed on it to allow for multiple first packet data items to be transferred simultaneously

Name	Semantics
string init_send()	Compute the token to be exchanged in the first packet
void init_recv(string s)	Receive the first packet token from the remote node
string generate_challenge()	Compute a challenge to force remote node reauthentication
bool test_challenge(string s)	Test the validity of the challenge sent from the remote node
string generate_challenge_response()	Compute the authentication information as a response to the challenge
bool test_challenge_response(string s)	Test the challenge response

Table 17: Important methods of *authentication*

methods *generate_challenge* and *test_challenge* are used to generate and evaluate an authentication challenge. The methods *generate_challenge_response* and *test_challenge_response* are generate and evaluate a response to a challenge.

This prototype supports a specific type of challenge-response authentication system based on DCCP. DCCP's initial authentication data consists of a 32-bit nonce. A challenge consists of a reauthentication request along with an XOR of the nonces exchanged at connection establishment time for verification. A challenge response from an end point consists of remote entity's nonce. This scheme is vulnerable to a variation of man-in-the-middle attack, but it is included here only for experimentation purpose.

A.8 Name Service

Source Files: dnsserver.h, dns_client.h

This prototype includes a primitive name service. Table 18 lists various messages exchanged between clients and DNS servers. They include messages for registering, unregistering and lookup names. The interface provides for registering a combination of both domain and service name. The service name allows for flexibility of the implementation of the service. The slots numbers need not be well defined, and different service could use different routes.

The function of the dns_client class is to provide a simple interface to the DNS function. It assumes that the FD corresponding to the DNS has already been discovered and specified.

Table 19 list the methods associated with this class. Figure **??** shows the typical usage of the

A.9 Miscellaneous

The implementation includes several classes that are necessary for completeness but not part of the high level design. In this section we discuss three such classes, viz., *counted*, *serializable* and *farads_debug*.

A.9.1 Counted Objects

Source Files: counted.h

Several objects, like slots, have an underlying implementation that is tied to an OS structure that needs to stay around, yet for transparency it would be clean if th object could be passed as a parameter or assigned to other objects of the same type. One can imagine an array of slots, for example. Such underlying objects are reference counted, and because this technique is used frequently, there is a class to support it: the *counted* class. For example, the underlying class of a *slot* is the *connected_socket* class, that is derived from the *socket* class, which is derived from the *counted* class. Counted provides three methods, summarized in Table 20.

Whenever a new reference is created to the underlying object, by an assignment or initialization, for example, the reference count is incremented. Whenever a reference is removed, by assign-

Type	Value	Contents	Semantics
DNS_REGISTER_REQ	1	type, name, RT, FD	Register name to (FD, RT) mapping
DNS_REGISTER_REPLY	2	type, status	Register operation status
DNS_UNREGISTER_REQ	3	type, name	Unregister request
DNS_UNREGISTER_REPLY	4	type, result	Unregister operation status
DNS_RESOLVE_REQ	5	type, name	Name lookup request
DNS_RESOLVE_REPLY	6	type, result, RT, FD	Result of name lookup result

Table 18: Various DNS messages

Prototype	Semantics
int register_name(string domain_name, string service_name, string rendezvous_string, forwarding_directive *fd)	Register a name to FD mapping with the DNS. Return the status.
int unregister_name(string domain_name, string service_name	Invokes talktodns() to unregister a mapping. Return the status.
int resolve_name(string domain_name, string service_name, string &rendezvous forwarding_directive **fd)	Invokes talktodns() to lookup a name. Return a rendezvous string and a destination FD.
void talktodns(DnsMessage &req, DnsMessage &reply)	Establish an association with the DNS, send query contained in req and return the response in reply.

Table 19: Class dns_client methods

Prototype	Effect
void add_reference()	Increment the reference count for this object
bool remove_reference()	Decrement the reference count, and return true if the count has become 0.
int references()	Return the current reference count

Table 20: *Counted* operations

ment or destruction, the reference count is reduced, and if the count is zero, the removing object deleted the reference-counted object. Slots use this to reference *connected_socket* objects, and *connected_association* uses this mechanism to share a queue of delivered messages and a state variable.

A.9.2 Serialization

Source Files: serializable.h

Several classes are serializable, meaning they inherit from the purely abstract *serializable* class. This class defines an interface for packing and unpacking objects for transport over the network. The functions are given in

These classes encapsulate the work of moving objects to and from the network, or even across a local socket. They are used by both *slot s* and *connected_association s* during their protocol exchanges and *forwarding_directives* are also required to be serializable.

A.9.3 Debug

Source Files: farads_debug.cc, farads_debug.h

farads_debug is an optional facility for use by applications and the various *libfarads.a* source files. The debug object is assumed to be available through well known global variable, *farads_dd*. Each debug message is encapsulated with a check for the existence of the debug object and a debug-level check. The debug messages are written to *FOUT* which has C++ *cout* semantics. Applications enable the debug messages of a certain level by instantiating a *farads_debug* object, assigning it to *farads_dd*. Three debug levels are available, viz., *DLO*, *DMID*, and *DHIGH*, and they can be specified using the *farads_debug* constructor.

Prototype	Effect
int serialize(void *b, int len)	Put a transferable copy of this object into the buffer. Return the number of bytes written.
int unserialize(void *b, int len)	Get a transferable copy of this object from the buffer. Return the number of bytes read.
int serialized_size()	Return the size of the object when serialized successfully.

Table 21: Serialization operations

A.10 Example

Figure A.10 shows the implementation of a echo server in FARADS. This example program shows how to use slots and connected associations. A companion program “ping” makes an active open to the slot this program allocates and send a message, to which this program responds with the message “pong”. This is a simplified version of the *pong* application included with the source distribution.

A few points to note from the example are as follows: (1) the program assumes that faradd kernel is always able to assign the application the slot it requests (2) the companion program knows which slot number to use (3) no source FD is specified explicitly for the connected association. Source FD is computed in the background as mentioned in Section A.4.

A.11 Summary

This appendix discusses the various internal data structures and interfaces in more detail. This prototype could be used to further experiment with FARADS concepts or be as a generic infrastructure for novel architectures. The implementation is easily extensible and customizable, and the interfaces are well defined. Further, the code is relatively well documented. We encourage the reader to tinker with this prototype.

```
#include <iostream>
#include "farads.h"

// Allocate a slot given by the first argument (or 3 if no
// such argument is given) and listen on it for the number
// of association given in the second argument (or one if
// no such argument is given). For each message, print the
// message and reply with a message containing "pong" and
// close the association.
int main(int argc, char ** argv) {
    try {
        // a buffer to read incoming data
        const int bufsize=4096;
        char buf[bufsize];

        // get the arguments
        int slot_num = (argc > 1) ? atoi(argv[1]) : 3;
        int iterations = (argc > 2 ) ? atoi(argv[2]) : 1;

        // Allocate the slot
        slot s(slot_num);

        // Make the passive association to listen on
        connected_association aa(s, "pong rv");

        // Process each association.
        for (int i = 0; i < iterations; i++ ) {
            connected_association a = aa.listen();
            int len = a.recv(buf, 1024);
            string pong = "pong";

            if ( len == -1 ) fatal("recv");
            buf[min(bufsize-1, len)] = '\0';
            cout << "Got request: " << buf << endl;
            if ( a.send(pong.data(), pong.length()) == -1)
                throw farads_exception("Send failed");
        }
    } catch (const farads_exception &e) {
        // Catch errors in all of the above
        cerr << "failed: " << e.what() << endl;
        exit(20);
    }
    exit(0);
}
```

Figure 20: A simple example - *pong* - that uses *connected_association*