The Network Architecture of OSF/1 AD Version 2

John LoVerso
Paul Roy

OSF Research Institute
{loverso,roy}@osf.org

February 2, 1994

1 Introduction

Between 1991 and 1993, the OSF Research Institute developed the OSF/1 AD operating system for multicomputers. Version 1 of OSF/1 AD became operational in 1992 and has served as the basis for Paragon OS as shipped on the Intel Paragon. Version 1 is also the basis for other vendors' technologies.

Presently, the OSF Research Institute is engaged in the design and implementation of OSF/1 AD Version 2. This document describes the architecture of the Version 2 networking code to support a well performing and scalable system for a parallel MPP or cluster environment.

Background information on the overall OSF/1 AD architecture (e.g., a Mach microkernel runs on every node, servers provide UNIX functionality, etc.) may be found in section 2 of the ADFS architecture document [4].

2 Key Requirements

There are several key requirements of the networking subsystem. This is a summary of those requirements that this architecture addresses:

- *Fully Functional.* Support for all base OSF/1 networking functionality, including Internet and UNIX domain sockets, STREAMS, pipes, FIFOs, etc., as well as full OSF/1 API compliance.
- *Single System Image.* All networking functionality must be transparently accessible from all nodes. Additionally, the multicomputer must appear as a single Internet host.
- *Network Interfaces*. Support for multiple physical network interfaces across multiple nodes.
- *Scalability and Performance.* The system must scale to 1000's of nodes, necessitating distribution of functionality so that messages and copies are avoided (e.g., by queuing data in the "most performant" place).
- *Configurable Compute Nodes.* To minimize memory consumption (possibly at the expense of networking performance), it must be possible to configure nodes with a minimum of network functionality.
- *Failure Containment.* The failure of a server involved in network operations should limit the loss of resources to just the connections being serviced by that server.

This research was supported in part by the Advanced Research Projects Agency (ARPA) and the Air Force Material Command (AFMC).

3 Two-Phase Implementation

The networking functionality described in this document will be developed in at least two phases. The first phase, to last through 1994, will put in place a solid base architecture and provide full single system image networking support. In addition, many issues of scalability and performance will be addressed during this phase.

A second phase will build on the first phase to provide further scalability and performance enhancements. For instance, it is expected that the first phase will support network interfaces on any node by servicing them from just one server, with the second phase removing this restriction. Each section in this document will discuss the functionality to be provided in each of the two phases as appropriate.

Note that the second phase activities will not be planned in detail until the first phase nears completion. It is possible some vendors will choose to provide enhanced functionality on their own, or perhaps engage in a joint work arrangement with the OSF RI.

4 Background - Base OSF/1 Networking Functionality

This section provides background on the base OSF/1 networking functionality, and hence serves as a template for the areas needing attention in AD Version 2.

4.1 Sockets

Sockets are a framework for accessing network protocols. This framework is comprised of several layers, each utilizing well-defined interfaces. The framework supports both stream and/or datagram characteristics. It also allows endpoints to be bound to a name and/or connected to another endpoint. A socket can be used to listen for new connections. A family of protocols using the socket framework is called a *domain*, although the terms address family and protocol family are also used. Sockets are named by a *sockaddr*, the contents of which are domain specific.

4.1.1 UNIX Domain

The UNIX domain allows communication between processes within the same system. A sockaddr for this domain contains a pathname. Binding a UNIX domain socket makes it accessible through the file system name space. Figure 1 shows the data structures that exist in OSF/1 for a bound UNIX domain socket.

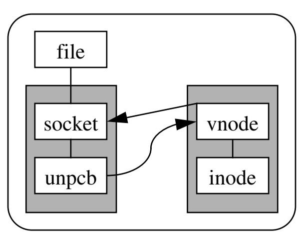

Figure 1: A bound UNIX domain socket.

The unpcb structure is the protocol-specific control block for a UNIX domain socket. Its unp_vnode pointer points at the bound name. The vnode contains a v_socket pointer used to reference back to the socket.

UNIX domain sockets utilize the ability of sockets to pass "access rights" to provide a means to pass open file descriptors from one process to another. Although the OSF/1 implementation of this feature is fully functional, for historic reasons there is very little use of this mechanism. Further, there is no standards-level requirement for this mechanism.

Connections in stream-oriented UNIX domain sockets result in new sockets for bidirectional data transfer as shown in Figure 2. The connection will last for the lifetime of the pair of sockets.

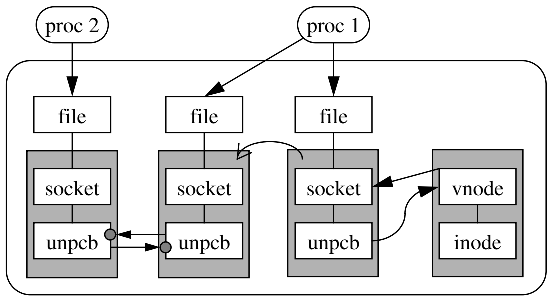

Figure 2: Connected UNIX domain stream sockets.
A new connection from the client socket on the left to the listening socket on the right results in the creation of the socket in the middle, which is bidirectionally crossed with the client socket.

Datagram-oriented UNIX domain sockets use unidirectional connections to send data as shown in Figure 3. Such connections may be a lasting association or exist only for the duration of a single send operation.

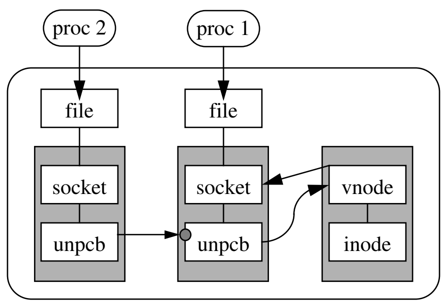

Figure 3: Connected UNIX domain datagram sockets.
The connecting client socket unidirectionally attaches to the server socket's unpcb. Data can only flow to the server.

4.2 Internet Domain

Internet domain sockets support communication between endpoints on local or remote systems using the Internet protocol suite. Binding an Internet domain socket attaches it to a port in a protocol specific name space. Internet domain sockets may also be bound to wildcard addresses when listening for new connections. The data structures involved with Internet domain sockets are similar to those in Figures 1, 2, and 3, except that there is an Internet protocol control block and a protocol-specific port space instead of an attachment to a vnode/inode.

The Internet domain maps the socket datagram type onto the UDP, whereas the stream type is mapped onto the TCP.

4.3 FIFOs

A FIFO is a named pipe, accessed through the file system name space. The FIFO special file is persistent, but the FIFO it refers to is only usable when it is open for reading and writing.

In OSF/1, the underlying implementation of FIFOs uses a pair of connected, unbound, UNIX domain sockets. Since FIFOs are still unidirectional streams, one socket is disabled for receiving while the other is disabled for sending. Figure 4 shows the data structures that exist in OSF/1 for a FIFO.

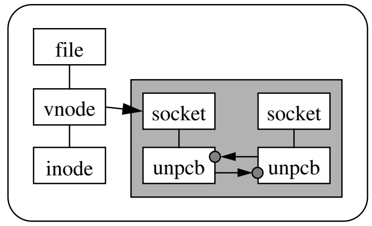

Figure 4: Opened FIFO
The first open of a FIFO special file will create a pair of connected UNIX domain stream sockets. The vnode's v_socket field holds a pointer to the one of the sockets.

4.4 STREAMS

The STREAMS framework provides a data communication infrastructure. STREAMS are accessed through the file system name space as special file types. A new stream is created by opening a STREAMS character special device. The stream is accessed via its stream head (sth) structure, which is the linkage responsible for converting operating system requests into STREAMS messages.

Most STREAMS devices are used as *clone* devices. Such devices have a single special file that results in a free minor device allocated for each open. The clone device open operation is intrinsic to the character special device handling of *specfs* rather than being a part of STREAMS. A hash list of allocated minor numbers is maintained for each device major number. A clone open always returns a new, "anonymous" vnode that is initialized from the vnode/inode pair of the special file.

4.4.1 Pipes

Pipes are implemented by a connected pair of open STREAMS pipe devices, created off the clone device /dev/pipe. The connection is made by a special I_PIPE ioctl() against one of the steams, giving the other as a parameter. The data structures that exist in OSF/1 for a pipe are shown in Figure 5 (the vnode/inode pair for /dev/pipe, from which these vnodes were cloned, is not shown).

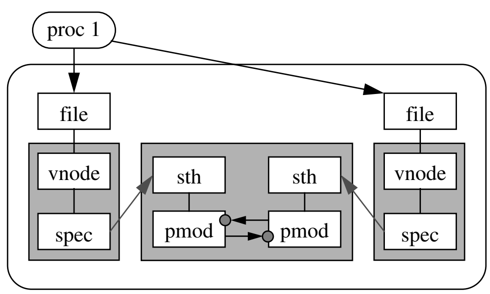

Figure 5: A new STREAMS Pipe
The I_PIPE operation directly crosses the queues of the pipe modules (pmod).

A pipe starts as an unnamed object. However, it is possible to name the pipe at a later time by using the fattach() system call. This operation makes use of the File-on-File Filesystem to mount the pipe file descriptor on a pathname.

4.4.2 TTYs and PTYs

The terminal subsystem in OSF/1 is implemented by use of the STREAMS framework. Since STREAMS modules are used, the relevant data structures, from a distribution standpoint, are similar to those for pipes when viewed above the pipe module. Whereas ttys utilize a device driver to access the hardware serial port, a pseudo-tty (pty) is composed of just a paired set of STREAMS devices, each with a separate major device by sharing the same minor device space. These are termed the master and slave sides of the pty. Additionally, a non-STREAMS device is also available. It provides a BSD-style master side for the pty to provide backwards compatibility to programs expecting such behavior, and works along with the STREAMS slave side. While not a STREAMS device, it is intimately aware of the STREAMS implementation of the slave side of the pty.

5 Distribution Mechanisms

Distribution of the OSF/1 networking functionality in AD Version 2 will focus around two core elements: a distributed socket abstraction and the distributed file system. This section describes those mechanisms and Section 6 describes how these elements are utilized by each of the networking components.

The networking functionality will leverage other technology being developed for AD Version 2. Operations that result in the generation of signals will do so via an appropriate vproc operation [3]., and distributed state, as well as references to remote objects, will be kept synchronized via the *token module* [2].

5.1 Socket Handles

A *handle* [4] will be used for all remote references to a socket. This handle will be a tuple of the form (server id, object id), with 32 bits of server id and 32 bits of object id. The handle will be created at the time a server needs to pass a socket reference to another server. The object id will be the virtual address of the socket distribution data structure inside of the managing server. In almost all instances, the validity of a socket handle will be assured by a corresponding *existence* token.

The notion of an *indirect handle* will also be utilized. This handle will be a tuple of the form (server id, key) and provides a handle *hint* for an object. As such, this type of handle never has an associated existence token. The key field guarantees that an attempt to reference this hint will find whether the object is extant. It also guarantees that the handle will never find the wrong object, which is a potential problem with normal socket handles since the socket data structure may get reused for a new socket.

An indirect handle will typically be used when an object reference needs to be exported into another subsystem. This is intended to replaced those instances in OSF/ 1 where a socket is referenced by a vnode. The file system does not access the socket itself, but instead merely needs to pass the socket reference to other servers. In this case, an indirect socket handle is a cheaper form of reference in that it does not involve the overhead associated with an existence token.

The allocation of an indirect handle involves allocating an object bucket, which contains a pointer to the object the handle will represent, as well as a key value. The key is assigned the new server-wide indirect generation number every time the bucket is allocated and will be used to record the bucket in a hash table. It is also the value used in the key field of the indirect handle.

5.2 Distributed Sockets

OSF/1 AD Version 2 will implement a distributed socket architecture that allows socket clients to reside on nodes other than where the physical socket lives. This is accomplished via a Virtual Socket (vsocket) layer that introduces a level of indirection much in the same way that vnodes have abstracted the notion of files. This will be similar in nature to the layer used in OSF/1 AD Version 1 [6].

The existing **socket** data structure will be split at the vsocket layer. Client access to a socket is performed by invoking operations on a **vsocket** data structure which, in turn, invokes operations on the underlying socket implementation, either by accessing an actual local socket or by sending a message when the socket is remote. State for remote sockets is maintained in a distribution layer. The server where the socket resides will have a **psocket** data structure that retains information about the distributed nature of the socket, including all of its clients. A remote client will have a **dsocket** data structure that maintains state and a handle to the remote psocket data structure. It will be possible to interpose a psocket into an existing vsocket/socket pair. Figure 6 shows these new structures.

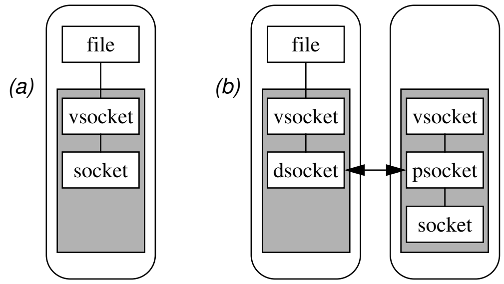

Figure 6: Virtual Socket Layer

Local (a) and remote (b) sockets represented by a vsocket.

The token module will be used by the distributed socket implementation to track which clients possess handles for a particular socket. Specifically, a remote client's dsocket will possess an *existence* token for the actual socket, and that token must be revoked (causing the dsocket tear down on the client) before the socket can be torn down.

In the initial implementation, all queued data associated with a socket will reside with the physical socket. A send operation by a remote vsocket will pass data directly across from the dsocket into the psocket, placing that data in the sockbuf of the socket. Similarly, a receive operation will result in an RPC into the psocket, which will call into the socket and block if no data is immediately available. The case for a select() operation is slightly different, since it involves the queuing of callbacks on multiple file descriptors. This will require an RPC into the psocket to queue a request for the select callback, which in turn will become a second RPC from the psocket back to the dsocket to wake up the sleeping select thread. Since the nature of the select implementation in OSF/1 currently involves three exchanges between the file layer and the socket, care must be taken to insure that these don't turn into three separate RPCs.

A planned performance enhancement for phase two will enable migration of a socket's *storage site* (the receive side socket buffer) to other servers (e.g., to the server managing the vsocket of the sole or *primary* reader). A server with access to such a socket will cache the location of the socket's storage site with the aid of a token; when the storage site changes the token will be revoked, forcing the server to reacquire the token and hence update its cached knowledge of the storage site location. The motivation for this scheme is that in the typical case of a TCP connection, there is exactly one client of the socket. This architecture would allow the storage site to be placed with the client's dsocket. The psocket would push incoming data directly onto the dsocket. A client's receive or select operation would no longer go remote to the psocket; instead, it would be processed locally by the dsocket. The advantage is that the latency of the data transfer between psocket and dsocket is asynchronous to a read operation. The performance improvement will be best seen by clients whose method of operation typically involves:

- computation between receives, where data is likely to arrive before a receive blocks;
- non-blocking I/O, so that an RPC can be avoided;
- loops alternating between select() and read(), such as in most Internet daemons.

Finally, phase two will introduce an extension to the dsocket implementation so that it can refer to multiple psockets. This is a requirement for the implementation of multiple network servers where an address family allows a socket to be bound to a multicast or wildcard address. Thus, a single vsocket might need to coordinate incoming connection requests from multiple physical sockets, each bound on one network interface.

5.3 Distributed File System

The OSF/1 AD distributed file system (ADFS) [4] plays an integral role because some types of network objects are accessible through the file system name space. Each type of object uses the file system in a different way and requires a different distribution strategy. Therefore several different mechanisms will have to be developed to handle all the different cases.

In the case of a bound UNIX domain socket, ADFS will provide the cross-nodes linkage so that the vnode representing the name that the socket is bound to may actually be a client-side vnode connected to the corresponding server-side vnode where the name's inode resides. This allows the name of a socket and the socket itself to be managed by different servers. The v_socket field of the server-side vnode will be used to hold an indirect socket handle for the socket it names. Figure 7 is a modification of Figure 1 that reflects this. Note that an ADFS interposer structure is used purely to hold a reference to the name and that no data transfer ever takes place via the file system.

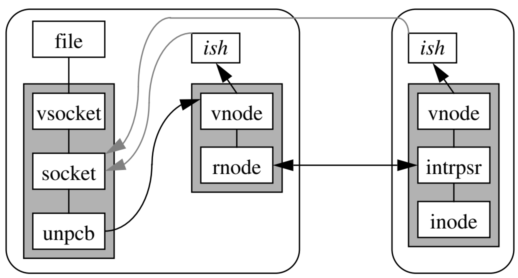

Figure 7: UNIX domain socket with cross-node binding

The unpcb holds a pointer to the local vnode, which is an ADFS rnode that holds a remote reference to the vnode of the newly created VSOCK inode. Both vnodes hold the indirect socket handle (ish) that references the socket.

Similarly, the pipe comprising a FIFO can be managed by a server separate from the file server managing the name of the FIFO. This is unlike the previous case, however, in that the name in the file system is persistent while the object is transient. Also, the file system is the only conveyor of data. A FIFO shares the property of remote special device files, in that there need to be separate paths for the data and attributes within the ADFS linkage. Figure 8 is a modification of Figure 4 that reflects this, and is described in detail in Section 6.2.

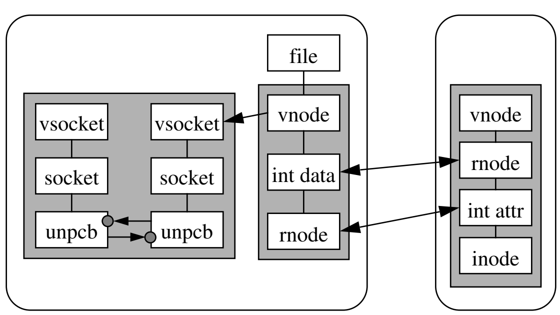

Figure 8: FIFO remote from inode

The FIFO exists on the left server with the server on the right managing it's name. A new open of the file will result in VFIFO specific code directing the new client to the vnode on the left.

Finally, since STREAMS are devices accessed via character special files, they will utilize a mechanism similar to the remote device support in ADFS. A particular stream, however, will be created when it is first opened. Thus, the node managing it is dynamically assigned and will probably be on a server other than where the special file exists. Since all data transfer will happen through the vnode, ADFS will also be responsible for the cross server linkage of a stream after a remote fork. Figure 9 reflects this aspect as a modification of Figure 5.

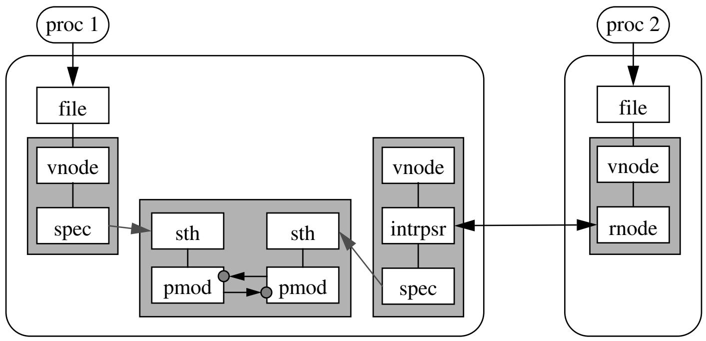

Figure 9: STREAMS pipe after remote fork()
The child process (proc2) has closed one end of the pipe and the parent has closed the other.

A particular ADFS concept that must be kept in mind is that when the special file is remote there are vnodes on both the client side and server side. However, these vnodes represent the same object and "collapse" together when the file system client is the same server as the one managing the special file. This is one reason why both vnodes in Figure 7 hold the socket handle. It is important to remember that ADFS will only create a vnode interposer when it is necessary. Thus, an interposer will get inserted transparently if it doesn't already exist at the time of the first remote lookup.

6 OSF/1 AD Version 2 Network Architecture

This section builds on Sections 4 and 5 to show how the base OSF/1 networking functionality is extended to a distributed environment.

6.1 Sockets

The virtual socket layer will provide the ability to refer to a socket in a remote server. Phase one of the implementation will include a restructuring of the base OSF/1 networking code to change socket references into vsocket references. A direct local socket will be handled by a pass-through vsocket, while a distribution layer will be available with the psocket-dsocket pair.

6.1.1 UNIX Domain

For considering the implications of distribution, the interesting operations on a UNIX domain socket are create, bind, connect, data transfer, and disconnect/close. A brief discussion of each follows.

Creating a UNIX domain datagram or stream socket will result in a vsocket and a socket being created and managed by the local server. A datagram socket is immediately useful via a send operation that contains a destination address. It may be connected to an address that will act as the default destination for send operations that do not specify one. Finally, it may be bound to a name to allow other sockets to connect to it. A stream socket will need to be fully connected before being used. This requires that it either be bound to an address and then listened on, or that it be connected to a destination.

Binding a UNIX domain socket attaches it to a name in the file system that must not already exist. Binding first causes namei() to be invoked, resulting in a vnode referencing the parent directory. Next, the bind code will allocate an indirect socket handle to represent the physical socket. The indirect handle will be passed in the vnode attributes structure so that the VOP_CREATE call can atomically create the VSOCK file. The file server will store this handle in newly allocated memory, referencing it in the vnode via the pointer-valued field that previously held the v_socket pointer in the vnode. The indirect handle will also be cached by any client side vnode's representing the file, including the local vnode in case that the file is initially on a remote file server. Allowing the indirect handle to be cached within each client vnode is valid since it only represents a hint rather than a guarantee that the object exists. Since the VSOCK file cannot be reused after the socket is destroyed, any cached indirect handles will become meaningless and the storage containing them will be freed when the vnodes that reference them are reclaimed.

While the lack of the use of an existence token for the indirect handle allows the avoidance of any vsocket specific code on a file server for token manipulation, it does open certain race conditions. In particular, one client could be doing a file lookup on the VSOCK file while the socket was being torn down. The first step of tearing the socket down will be to invalidate the indirect socket handle, which might have already been returned with a successful lookup. A subsequent lookup of the indirect handle by the client will contact the server managing the socket, only to discover that the handle is invalid. Thus, the client will return a failure indication, which would typically be ECONNREFUSED for the case of connect().

While the mechanics of connecting a UNIX domain socket depend upon the type of socket involved, the lookup portion of the operation is the same. The server managing the client socket does a namei() to lookup the vnode representing the VSOCK file. As part of the lookup process, ADFS might need to create a local, client-side vnode to represent the remote VSOCK file. The lookup implicitly includes the retrieval and local storage of the indirect socket handle from the server's vnode. Thus, the binding code will just be able to copy the handle from the vnode, after which the vnode is vrele()'d. From this point on, the UNIX domain socket will have no further interaction with the file system.

Once the indirect socket handle is available, the client socket uses it to retrieve a socket handle for the listening socket controlled by an existence token, after which the indirect socket handle is no longer used. The client socket then needs to complete the connection. The action is different, though, depending upon the type of socket involved. For stream sockets, a connection results in a pair of connected sockets, being the client socket and a new socket cloned off the listening socket. Thus, the client socket creates a new, local UNIX domain socket. It then invokes VSOP_CREATE() on the listening socket, causing the server managing the listening socket to create a vsocket which will be queued as a new connection until the listening process does an accept() system call. The queued vsocket will have a dsocket pointing back to the newly created socket in the client's server. The newly created socket can now exchange its receive and transmit queues with the client socket. The result is a full duplex connection such that data is transmitted by directly queuing it in the receive sockbuf of the paired socket. Figure 10 shows this final state.

For datagram sockets, the completion of the connection is simpler. The socket handle will be saved and used each time a datagram is to be sent. When a datagram is ready to be queued to the receiver's sockbuf, it will be passed remotely via a direct RPC to the socket handle using an operation specific to the UNIX domain implementation. This connection can be broken by the client making a connection to a different name. In this case, the handle for the connected socket is cleared and the existence token released.

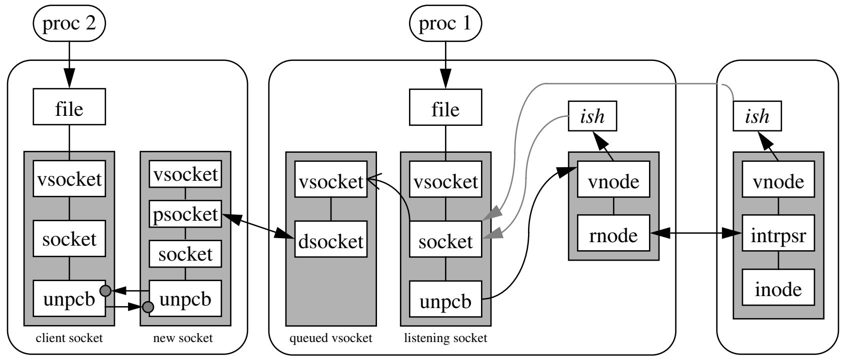

Figure 10: Newly connected UNIX domain stream socket

Process 1 still contains a fd for the listening socket. Process 2 has connected, creating the new socket in the server on the left. The new socket is remotely referenced by a vsocket queued on the listening socket. When process 1 does an accept() on the connection, the queued vsocket will have a file structure attached.

Since no open systems standard requires it, and it is unlikely to be an end-user requirement, OSF/1 AD Version two will not attempt to address the ability for UNIX domain sockets to pass file descriptors via access rights. If it becomes a requirement, the implementation would leverage the code necessary to export open files during a remote fork(). This involves interposing a psocket on the existing local socket if it doesn't already have one, and then creating a remote vsocket/dsocket pair. STREAMS provides a similar ability with I_SENDFD (see section 6.4.5).

6.1.2 Network Interfaces

The phase one architecture will only provide a single network server for the entire system. This server will manage the physical network interfaces attached to all nodes of the system. Those interfaces on remote nodes will simply utilize NORMA IPC to access the microkernel device interface. This restriction to using a single network server affects those network transports that utilize hardware interfaces (primarily Internet domain sockets).

Phase two will provide efficient and fully scalable access to network interfaces by allowing network protocols to execute directly on each node with an interface. One mechanism to accomplish this will be an additional multiplex in the vsocket layer to allow a single vsocket to refer to multiple remote sockets, each existing in a separate network server. This multiplex is needed to support a protocol binding to multiple network interfaces, such as an Internet domain socket listening for connections.

An additional performance improvement could be to distribute protocol processing across the servers managing client connections. This would alleviate the potential bottleneck associated with a network server executing all the protocol processing for its attached network interfaces. A mechanism to accomplish this would be to incorporate the packet filter enhancements done for MPF[1][5] into the OSF microkernel. This makes it possible to have fast packet filters that can handle per-connection (or per-endpoint) destinations. This would allow a connect in a transport protocol (TCP) to utilize a direct path from the microkernel holding the network interface to the server managing the vsocket, bypassing the server that is nominally servicing the interface. Once a connection has been established, the protocol state would be migrated from the network server into the server the user application is utilizing. A packet filter for this connection would be established for each network interface to push all protocol traffic for this connection to that specific socket. This would provide a data path to remote nodes where only a single operating system server is involved.

6.1.3 Internet Domain

Since Internet domain sockets are restricted to a single network server in phase one, and the file system is not involved, the support is simpler than that required of UNIX domain sockets. The creation of a new socket will first require the local server to lookup the server id of the network server. Then, a VSOP_CREATE is issued; this results in a local vsocket/dsocket being created to refer to a vsocket/psocket/socket on the server. Attached to the socket is the Internet protocol control block (inpcb) under which a protocol specific control block is used. All protocol operations will occur on the network server.

Phase two will provide support for multiple network interfaces (see 6.1.2). Since the Internet protocol suite allows a protocol to bind to a wildcard address, allowing the socket to accept connections from any of the local hosts's network interfaces, the dsocket implementation will need to be augmented. This will apply specifically to those sockets being used to listen for incoming connections; the dsocket layer will need to link to a psocket on each of the network servers. The token module[2] will be used to manage the list of socket handles for each of these psockets.

6.1.4 Other Protocol Families

OSF/1 AD Version two will not support other protocol families, although it is not expected to be a major effort to adapt existing protocol implementations such as the OSI stack into the vsocket architecture.

6.2 FIFOs

Since the FIFO is a persistent object in the file system, its name is always accessed by a single inode. The first server to open a FIFO results in the creation of the underlying sockets that implement the behavior. This architecture utilizes ADFS to place the behavior of the FIFO remote from the server managing it's name.

The initial open of a FIFO special file will occur by a client doing a lookup, resulting in the VOP_OPEN operation being invoked on the server managing the name of the FIFO. Instead of calling fifo_open() to locally to create a pair of sockets, the server invokes adfs_fifo_open() which, in turn, arranges with the originating client to invoke fifo_open(). Thus, the originating file system client will create the FIFO locally and thereafter manages it. The server managing the FIFO's name will stack an ADFS rnode on the vnode that will pass all data operations over to an ADFS interposer the client will have inserted under it's vnode. Under the rnode, the server will have an ADFS interposer for file attributes, since these refer to the FIFO's inode. The client will use an ADFS rnode to reference and cache the attributes from the server. The intent of this architecture is that all data operations will be processed on the node with the actual FIFO, while the actual attributes will occur on the node of the server managing the name. Figure 8 is a complete picture of an open FIFO.

Any additional opens of the FIFO special file will be intercepted by the ADFS rnode, which will pass them off to the node managing the FIFO. Thus, adfs_fifo_open() is only invoked once when the FIFO needs to be created initially. The rnode interposing on the inode is sufficient to name the location of the actual FIFO.

The server managing the FIFO will continue to use the existing OSF/1 FIFO implementation with little or no additional changes. In particular, the pair of UNIX domain stream sockets used to implement the pipe behavior will operate as pure, locally connected objects. A vsocket will exist for each socket, but the vsocket layer will never come into play for FIFO distribution. Where the OSF/1 implementation used a socket pointer help in the vnode as it's only FIFO specific data structure, this implementation will use just a vsocket pointer. Note that the field used to hold this pointer is the existing field in the vnode used to hold a pointer specific to the type of vnode (the same pointer will be used for an indirect socket handle; see section 6.1.1).

While utilizing the vsocket layer might suggest itself as an additional distribution mechanism for FIFOs, that would not come without some additional implementation cost. Furthermore, future plans of OSF/1 lean towards a FIFO implementation based upon STREAMS, much like the pipe implementation of earlier OSF/1 implementations has been superceded by one based upon STREAMS. Therefore, this architecture provides the basic foundation for distributed FIFO access, but will leverage the phase two work for STREAMS as the necessary infrastructure will be part of pipes utilizing intra-STREAM distribution.

Note that the existing FIFO implementation calls into the socket code at various layers. I.e., fifo_read() and fifo_write() call directly into the socket code and are able to directly use the vsocket interface (sosend becomes VSOP_SEND). The fifo_ioctl() and fifo_select() operations call the corresponding fileops for sockets, which can be replaced by the equivalent functions from the vsocket fileops.

6.3 White Pages Service

A *white pages service* will be implemented to provide a simple object registry independent of any other name space, including that of the file system. The service is intended to be general in that it can handle the registration of many different types of objects. This is accomplished by using object specific registration and lookup functions. It has not yet been determined whether objects registered in the white pages service will constitute a guaranteed object reference (via an existence token).

The initial consumer of this service will be STREAMS. Since it is a special file system type, STREAMS will use the white pages to record the location of the open stream that represents a particular minor device number, since the managing server of the stream can be assigned dynamically. Since STREAMS pseudo-devices can be opened by processes running on different nodes but must be managed by exactly one server, the white pages will be used during processing of non-clone opens to determine which server, if any, is already managing such a device. For clone device opens, the white pages will be used to broker the allocation of unique minor numbers (section 6.4.1).

In phase one, the white pages service will be implemented by a single instance in the system. Replication and additional distribution of the activities of the white pages service will be considered for phase two.

Figure 11 outlines the messages that would happen for the opening of an object whose location is discovered via a lookup in the white pages service.

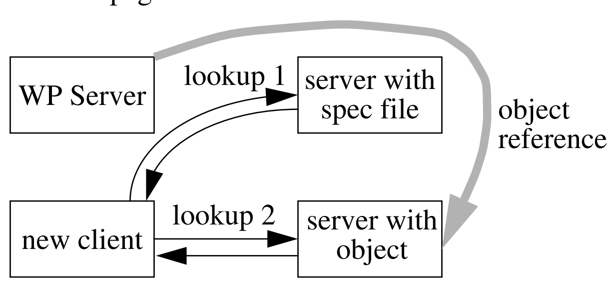

Figure 11: White Pages Usage

A client's lookup is processed by the server managing the special file. This server will use the White Pages to find the object, resulting in the client issuing a new lookup, retargeted to the server with the object.

6.4 STREAMS

The first phase of this architecture will provide the basic mechanisms so that STREAMS will function in a distributed fashion. This will involve the instantiation of streams on different servers, although each individual stream will be managed by exactly one server. Pseudo-devices comprised of more than one stream, such as ptys and pipes, will be limited to a having both streams managed by the same server. Phase two will allow the connection of arbitrary streams regardless of server to lift this restriction (see section 6.4.6).

6.4.1 Placement Policy

STREAMS pseudo-devices will be managed by the server where they are first opened. The node number in the special file will be unused and ignored. This has the advantage of providing load distribution in a multicomputer. STREAMS physical devices will be managed by the server where the device is attached, which will be identified by the node number in the on-disk inode.

The cornerstone of this strategy is that, unlike other devices in the system, a particular stream is uniquely identified by a major and minor device number, *without* the use of the node number.

For clone opens, the file server managing a stream's special file will be responsible for assigning a new minor device number. The stream that is named by this minor number will be created in the server of the file system client. The existence of this stream and the fact that it is named by the newly assigned minor number is recorded in the white pages service.

However, there are some complications to this scheme. It is possible for two different file servers to have clone special files that refer to the same STREAMS major device. In such a case, the minor device space needs to be correlated so that the same minor device is not allocated twice. Another example would be the case when a clone device exists but there also exists a special file with an explicit minor device number that is managed on a different server. If the clone device has allocated that minor number to an open stream, then an open of the explicitly numbered special file should refer to the existing open stream. This system will handle these cases by the use of a brokerage that passes out the right to allocate minor device numbers. This brokerage will allow each separate file server to keep their own set of minor device assignments, but will utilize the registration of a minor device in the white pages as a global allocation scheme. Hence, the white pages service will participate in the allocation of new minor numbers for clone devices.

6.4.2 Pipes

STREAMS pipes are pseudo-devices that implement an object with pipe characteristics. A full duplex pipe is created by welding together a pair of open STREAMS pipes by the use of the I_PIPE ioctl. Such a pipe is obtained by the pipe() function, which was previously a syscall but is now implemented by a library call that opens the clone device /dev/pipe twice, and then does an I_PIPE ioctl. /dev/pipe is a special device that has no minor units, since pipes are intrinsically unnamed (anonymous) objects.

Pipes created with the pipe() library call will have both ends of the pipe managed by the same server. This means that the weld point does not need to be able to span servers. Strictly speaking, any user can invoke I_PIPE ioctl on a pair of anonymous, unpushed, unconnected and unattached streams, but no applications are known to actually utilize this feature. Hence, the phase one architecture will not support the ability for STREAMS on separate servers to be welded together. Phase two will remove this restriction via an intra-stream distribution mechanism as discussed in Section 6.4.6.

Additionally, phase two will utilize the inter-stream aqueduct to push (migrate) each half of a pipe into the server managing it's consumer. This is intended to provide a significant performance enhancement by allowing the end of a pipe to be near their respective clients, which would then be able to directly queue data into the buffer on the other side of the pipe. A pipe would be the candidate for such a migration whenever it is found to be servicing a remote client, typically after a remote fork, as in Figure 9.

6.4.3 TTYs and PTYs

For physical serial interfaces, the node number of the character special device will be used to indicate which physical tty a special device refers to. Like other remote devices, this node number will also indicate the server that will manage the streams that constitute the tty. Ptys, being a pseudo-device, will have their streams placed dynamically much like pipes. The only caveat is that the minor device allocation via the white pages service will represent the minor device space of all three major device numbers related to the pseudo-terminal, the slave and the two masters.

In the target environment, pseudo-terminal performance is especially critical. The placement policy for STREAMS will attempt to place a pty local to the client of the master side, so that file system operations such as read, write, and select will carried out within the master's server. This is accomplished by deferring the creation of the streams modules for the pty until the master side is opened. Since the typical usage of a pty involves repeated loops of read, write, and select against the master side, these will not incur the performance penalties associated with remote operations.

Since the client of the slave side is typically forked off of the client of the master side, it is possible that these processes will be running on different nodes. If so, the slave-side client will access the pty via ADFS. In phase two, the master and slave sides of the pty can be split between different servers via intra-stream linkage (see section 6.4.6). The advantage of this mechanism is that both ends of the pty will then avoid remote interactions for operations such as select.

6.4.4 XTI

OSF/1 contains an implementation of the X/OPEN Transport Interface (XTI) for STREAM. This includes a module called xtiso, which allows a STREAMS protocol to link with the socket network interface layer. Although this feature is fully functional in OSF/1, no existing software supplied by OSF utilizes it.

The basic distribution mechanism to be used for xtiso will be similar to that used for FIFOs. An xtiso module will be managed by the same node as the network server where its socket lives. Since a STREAMS stack exists above it, remote clients can access the XTI interface via ADFS. This work is planned for phase two.

6.4.5 I_SENDFD

The I_SENDFD ioctl sends an open file descriptor to another process via a stream. This requires that the file's handle be passed in a STREAMS message. This work will leverage the mechanism to export file handle information that is also needed by remote fork processing.

6.4.6 Intra-STREAM distribution

Phase one has a restriction that an individual stream cannot be connected to a stream residing in a different server. This prevents the arbitrary use of the ioctl()s like I_LINK, I_PIPE, and I_PLINK, which link together individual steams. In phase two, STREAMS will be enhanced to include implementation of an intra-stream distribution mechanism, taking the form of interposition on STREAMS queues. This will allow the arbitrary interconnection of STREAMS modules, including the ability of a single stream to span server boundaries. Such a mechanism can improve performance by allowing, for example, data to be queued in the server where the data is being read. An example of this mechanism is shown in Figure 12, which is a modification of Figure 9.

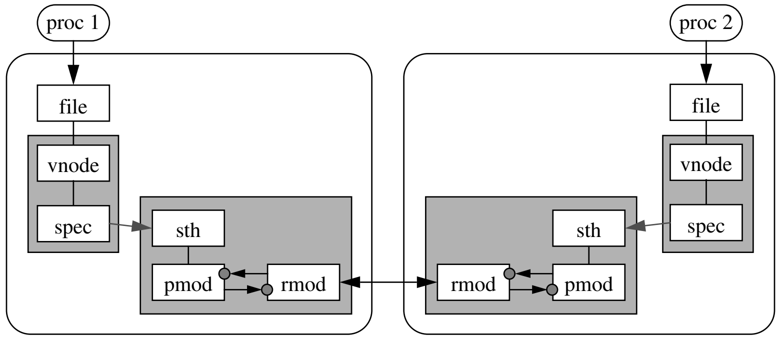

Figure 12: Intra-STREAM aqueduct

The remote module (rmod) links a local STREAMS queue with a remote STREAMS queue. In the pictured case, it is being used between two welded pipe modules; however, rmod is not specific to

6.5 Support for fattach()

The file-on-file-mount code (FFS) provides a hook that allows the fattach(3) function to exist. This allows an arbitrary open file, device, or STREAM to be rebound in the file system under a new name. This has considerable implications to pipes, which can now become bound, named objects after the fact. FFS must record the object handle under the name, since there is no requirement that they be co-located in the same server. This work will leverage the mechanism to export file handle information that is also needed by remote fork processing.

6.6 Minimal Client-Side Memory Consumption

In support of applications that require minimal memory consumption on a compute node, OSF/1 AD Version 2 will be configurable through the use of compilation switches. The ability to configure out unneeded parts of the system will be provided. A server configured in such a manner is termed a *minimal* server.

This support requires the concept of a network proxy server. For any server configured to trade off the performance of having STREAMS and UNIX domain sockets managed locally, another server will be designated as its proxy. Thus, any object creation attempted in the minimal server will result in the object being created in it's proxy, with the necessary vsocket or ADFS plumbing in place to reference the remote object. This scheme requires that all servers provide at least the client side of both the file system and vsocket layer.

The system also supports compute nodes that do not have a local server. In this case, the services for a process are provided remotely by having system call exceptions sent to a remote server via NORMA IPC.

6.7 Failure Containment

When a server fails, distributed state associated with the objects the server is managing or has reference to must be cleaned up. This will be accomplished by associating object references (and corresponding local data structures) with their server id's through a table local to each server. Each server id will map to a single Mach port, and the system will rely on a dead-name notification for this port to indicate when the server no longer exists. At notification time, the table will be traversed and the data structures associated with this server reclaimed.

6.8 Cluster Computing Model

At almost no cost over the support for Network Interfaces (see Section 6.1.2), a process in a cluster will be able to always use the local network interface for outgoing traffic. It will still be at the whim of the network and/or remote host for which interface is used for incoming traffic.One must avoid receiving *n*-copies of an individual packet from each of the *n* nodes in the system with network interfaces on network **N**. One viable approach is to assign each network interface its own network address (IP address in the case of Internet protocols). This leads towards a model where a cluster is built as a logical gateway to a private virtual subnet. Each node will advertise access to cluster resources via standard routing mechanisms for the particular network. Addresses in this logical subnet may be assigned on a per-node or per-endpoint basis. The latter provides a finer granularity of location transparency in the event that an endpoint moves within the cluster (e.g., due to process migration).

Note that support for cluster-style computing is not part of the current OSF/1 AD Version 2 plan.

7 References

[1] Maeda, C., Bershad, B. Protocol Service Decomposition for High-Performance Networking. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles.*

[2] Noveck, D. Token Module for OSF/1 AD Version 2. OSF Research Institute internal document. December 1993.

[3] Patience, S., Rabii, F. The Design of the Process Management Component of OSF/1 AD Version 2. OSF Research Institute internal document. November 1993.

[4] Roy, P., Noveck, D., Netterwala, D. The File System Architecture of OSF/1 AD Version 2. OSF Research Institute internal document. August 1993.

[5] Yuhara, M, Bershad, B., Maeda, C., Moss, J. Eliot B. Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages. In *Proceedings of the Winter 1994 USENIX Conference*.

[6] Zajcew, R., Roy, P., Black, D., Peak, C., Guedes, P., Kemp, B., LoVerso, J., Leibensperger, M., Barnett, M., Rabii, F., Netterwala, D. An OSF/1 Unix for Massively Parallel Multicomputers. In *Proceedings of the Winter 1993 USENIX Conference*.