Protocol Discovery in Multiprotocol Networks*

Russell J. Clark
Mostafa H. Ammar
Kenneth L. Calvert

GIT-CC-94/42

August 4, 1994

Abstract

Multiprotocol systems can be an important tool for achieving interoperability. As the number of protocols available on such systems grows, there is an increasing need for support mechanisms that enable users to effectively access these protocols. Of particular importance is the need to determine which of several protocols to use for a given communication task. In this work, we propose architectures for a protocol discovery system that uses protocol feedback mechanisms to determine which protocols are supported. We describe the issues related to protocol discovery and present feedback mechanisms necessary to support discovery. We present a prototype implementation of a discovery system that supports next generation IP protocols.

Keywords: multiprotocol systems, protocol discovery, protocol feedback, IPng transition

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280
rjc@cc.gatech.edu

*This research is supported by a grant from the National Science Foundation (NCR-9305115) and the TRANSOPEN project of the Army Research Lab (formerly AIRMICS) under contract number DAKF11-91-D-0004.

1 Introduction

The communication network is quickly becoming a critical component of computer systems in both educational and commercial environments. The abundance of personal computers at commodity prices as well as the overwhelming publicity associated with the National Information Infrastructure is fostering a significant growth in the number of networked computer systems. While this growth is impressive, it is important to recognize that the value of networking to the end-user is limited by the degree to which the needed resources can actually be accessed after going *on-line*. Just because a computer is connected to a network, for example through an electronic mail service, does not mean the user can access the resources available on the Internet. To a great extent, this access limitation is a direct result of the incompatibilities between different network protocols[1].

As data communications evolved, many different protocols were developed to support developing technologies and to address varying user requirements. This evolution has led to a great diversity in protocols that cannot interoperate. In order to promote interoperability, several standards organizations have worked to define standard protocols (e.g., TCP/IP, OSI). However, it is now clear that no single standard protocol or protocol family will become the universal protocol supported by all networked systems. Instead, large numbers of systems continue to be installed that support any one of the standards or one of numerous proprietary protocols (e.g., IPX, AppleTalk, SNA). Even the Internet is no longer a single protocol network [12]. While TCP/IP remains the primary protocol suite, other protocols (e.g., IPX, AppleTalk, OSI) exist either natively or encapsulated as data within IP.

Developing network systems that support multiple protocols can simplify the introduction of new protocols, like IPng, and reduce the risk for network managers faced with the prospect of supporting a new protocol. This will result in a faster, wider acceptance of new protocols and increased interoperability between network hosts. It has recently been pointed out that the National Information Infrastructure will be a multi-supplier, multi-technology endeavor that will create difficult interoperability problems. This will require mechanisms for negotiating commonality between network systems [20].

In our research, we consider ways in which multiprotocol networking can be accommodated through the use of multiprotocol systems [4]. In previous work, we have shown how a directory service can be used to provide the configuration information necessary for a multiprotocol system to communicate [5]. In this paper we present our research addressing the problem of what to do when no protocol configuration information is available through a directory lookup or the information that is available is inaccurate. We call our approach to solving this problem *protocol discovery*. In the current work we analyze the protocol features,

[1] While some access limitations are policy rather than technology restrictions, such cases are not the focus of this discussion.

especially feedback mechanisms, necessary to perform protocol discovery and point out limitations in some current protocols. We also present some practical approaches to performing discovery and describe our experience in implementing a discovery system.

In the next section we present the background for this current work including related work and a description of the multiprotocol model. In Section 3 we present the protocol discovery approach to determining which protocols a multiprotocol system should use. We present our implementation experience in Section 4 and discuss the feedback mechanisms of several common protocols along with ways they could be enhanced to better support discovery. We conclude with a summary and future work in Section 5.

2 Background

The general problem we address is how to design network systems that can interoperate in a multiprotocol network. Our approach to achieving interoperability is to develop multiprotocol end-systems that can directly communicate with many different protocol configurations. In this section we review related work in this area and then present the model for our research.

2.1 Related Work

Recently, others have begun to research the issues involved in integrating protocols from different architectures. Ogle et al. [14] are developing a TCP/IP and SNA system that performs protocol selection below the socket level interface. Janson et al. [10] consider options for interoperability between OSI and SNA networks, and analyze the addressing issues arising when these protocols are combined in a single network. One approach to achieving interoperability in diverse systems is to provide a mechanism for hosts to exchange protocol information before carrying out the communication task. Two early examples of this approach are the Network Command Language described by Falcone [8] and the "meta-protocol" concept proposed by Meandzija [13]. Similarly, Tschudin describes a "generic protocol" in [19].

The recent work of Comer and Lin [6] describes the use of a technique called *active probing* to deduce characteristics of a TCP implementation. While their work focuses on discovering possible problems with a known protocol, the technique used is somewhat similar to the discovery process we perform.

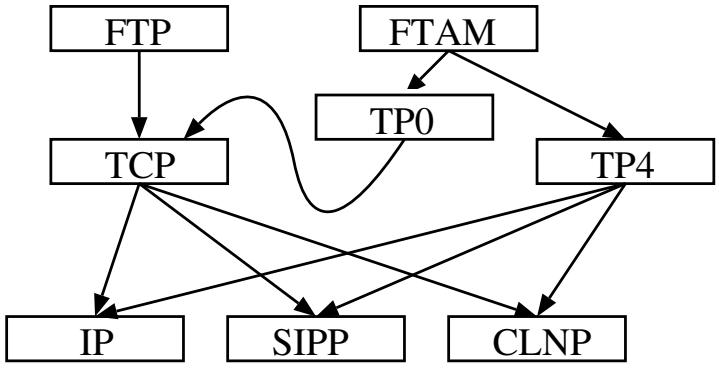

Figure 1: A Multiprotocol Graph

2.2 The Multiprotocol Model

A simple definition of a multiprotocol system is "a host that supports more than one protocol or protocol family". In this section, we provide a more formal definition and provide some examples of what we mean by a multiprotocol system.

In this work, we define a protocol as "a prior agreement among systems regarding the form and meaning of messages". A *protocol entity* (PE) is an object that implements a given protocol. With this protocol, a PE can communicate with another PE of the same type. In most cases, a PE of type a will use another PE of type b to transfer messages over to a's peer. This *uses* relationship between a and b gives rise to the common layering model that describes most network architectures. It is convenient to represent this relationship as a *protocol graph*. For instance, Figure 1 portrays a multiprotocol protocol graph. In this graph, each box or node represents a PE and each edge represents a uses relationship. Each instance of communication invoked by a user of a protocol graph involves a particular subset of protocols in the graph. We refer to this subset as a *protocol path* or simply a *path*. A path encompasses a fixed set of PEs, connected by the *uses* relation, that provides communication from the top layer PEs down to the bottom layer PEs.

A system supporting the protocol graph in Figure 1 provides a file-transfer service using seven different protocol paths. It supports the FTP application using TCP and the FTAM application using TP0 or TP4. This host supports three different network layer protocols. It supports the standard Internet protocol version 4, identified as IP. It also supports SIPP, one of the current proposed next generation Internet protocols (IPng) [7]. The OSI CLNP protocol is also supported along with the TUBA option [3] for providing TCP applications over CLNP.

Each of the graphs in Figure 2 provides a single protocol path. Figures 2a and 2f represent single stack architectures for the Internet and OSI protocols respectively. FTP using the SIPP IPng proposal is shown in Figure 2b. Figure 2g depicts a mixed stack architecture that provides the upper layer OSI services using the Internet

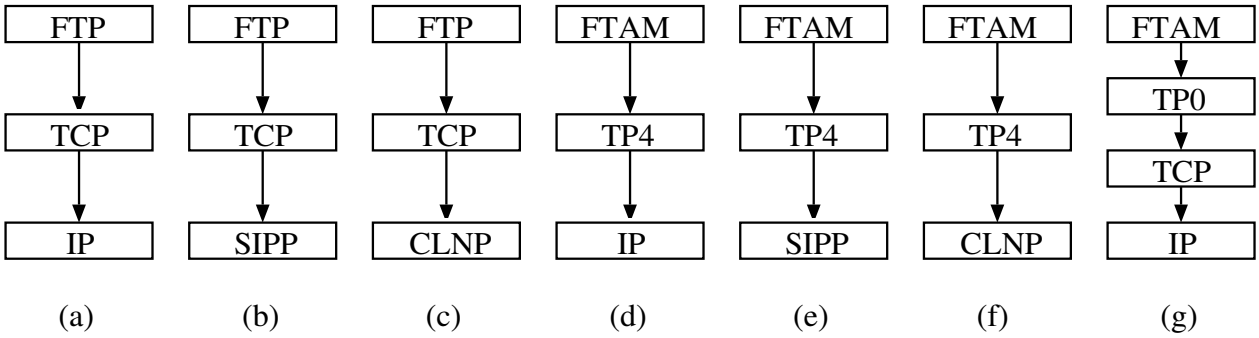

Figure 2: Single Protocol Graphs

protocols [18]. This is an example of a "transition architecture" for providing OSI applications without requiring a full OSI implementation. Figures 2d and 2e represent two other mixed stacks for providing OSI applications over the Internet. While these are less popular than 2g, they are indeed possible in our multiprotocol system. Figure 2c depicts a mixed stack architecture that provides the upper layer Internet applications using the OSI network protocol [3]. In addition to communicating with the two previous simple protocol stacks, the multiprotocol system of Figure 1 includes all the protocols necessary to communicate with these two new, mixed protocol stacks.

Unfortunately, in most current examples, multiprotocol architectures like that in Figure 1 are implemented as independent protocol stacks running on a single system. This means, for instance, that even though both TCP and CLNP may exist on the system, there is no way to use TCP and CLNP in the same communication. The problem with such implementations is that they are designed as coexistence (or so-called "ships in the night") architectures and are not integrated interoperability systems. We believe future systems should include mechanisms to overcome this traditional limitation. By integrating the components of multiple protocol stacks in a systematic way, we can interoperate with hosts supporting any of the individual stacks as well as those supporting various combinations of the stacks.

In order to effectively use multiple protocols, a system must identify which of the available protocols to use for a given communication task. We call this the *Protocol Determination* task. In performing this task, a system determines the combination of protocols necessary to provide the needed service. For achieving interoperability, protocols are selected from the intersection of those supported on the systems that must communicate. In previous work [5], we presented mechanisms for using directory services to assist in protocol determination. In the next section we describe an alternative approach called *protocol discovery*.

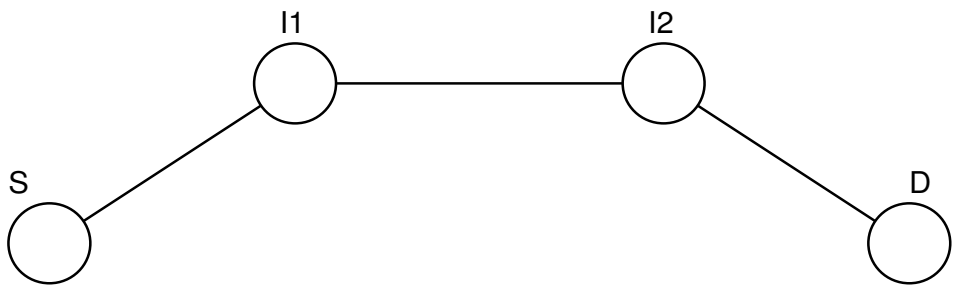

Figure 3: A Sample Network

3 Protocol Discovery

The idea behind protocol discovery is to use the features of networks and the network protocols themselves in determining which protocol paths are available to support a given communication task. This is done primarily by attempting to communicate using different protocols and monitoring the attempts to see what can be learned about the network configuration. Other sources of information can also be useful. A multiprotocol host could keep a cache of configuration information about hosts recently heard from. A host could also passively listen to other conversations on the network, especially a broadcast network, and learn about the protocols supported by hosts on the network. In protocol discovery, we use as much of this information as is available to the multiprotocol system for determining which protocols are supported on the remote system.

Here we present the protocol discovery concept, beginning in the next section with a simple example. After the example we give a formal description of the discovery task and present our proposed discovery architectures.

3.1 An Example of Protocol Discovery

Consider an initiating host supporting a multiprotocol graph such as that of Figure 1. This is the source or initiating host, labelled *S*, in the network with topology presented in Figure 3. The user of this host is attempting to communicate with the destination or responding host *D*. These hosts are physically connected via two intermediate network layer routers labelled *I1* and *I2*.

In order to perform a file transfer, the user must first determine which, if any, of the seven distinct local protocol combinations are supported on the remote host. The task is to determine which protocols supported by *S* are also supported by *D*. It is also necessary to determine the network layer protocols supported by both *I1* and *I2* as well as *S* and *D*.

In this case, without prior knowledge of the remote configuration, the user determines the protocols to use based on the feedback provided from the applications. The user proceeds by attempting a connection with one of the two applications and monitoring the way this attempt fails or succeeds. A successful connection indicates that the current application could be used. A failed connection indicates one of several possible problems.

Suppose for instance that host *D* supports the single OSI stack of Figure 2f. If the user decides to first try the FTP application with TCP and IP then this attempt will subsequently fail[2]. Table 1 lists the feedback provided to the user for various protocol incompatibilities[3]. In this case, since no compatible network layer is found, the user will receive the *Connection timed out* message. Since this message does not provide information about the actual cause of the failure, there is little to assist the user in choosing the next protocol combination to try. If the user continues by trying FTP with the two other options, both will fail. However, the attempt with the FTP/TCP/CLNP combination will fail with the *Connection refused* message. Based on the information in Table 1, the user can determine that CLNP is supported on the remote system. Now, when the FTAM application is tried, the user can choose the protocol combination that includes CLNP. In this example, the FTAM/TP4/CLNP option is tried and the communication succeeds.

Compatibility Problem	FTP Error Message	FTAM Error Message
Network congestion/partition	Connection timed out	Timer expired
Remote host off-line	Connection timed out	Timer expired
No compatible physical layer	Connection timed out	Timer expired
No compatible network layer	Connection timed out	Timer expired
No compatible transport layer	Connection refused	Timer expired
No compatible application layer	Connection refused	OSI service tsap#259 not found

Table 1: FTP and FTAM Error Messages

The "guided" protocol selection carried out in the above scenario is what we are interested in with protocol discovery.

3.2 The Protocol Discovery Task

Here we provide a more formal description of the protocol discovery problem. Consider the protocol path i, represented as p_i. At the beginning of protocol

[2]The first type of failure that might be encountered is the failure to find a network address for host D in the format required for the selected protocol. In order to simplify the example, the discussion here assumes that addresses are available for all the protocols attempted.

[3]The error messages listed are for the FTP program from Sun OS 4.1.3 and FTAM program from the ISODE version 8.

discovery the paths currently supported on the multiprotocol system S are known. That is, the members of the set $p_i \in S$ are known. Given this, it is necessary to determine for all $p_i \in S$ whether $p_i \in D$ or $p_i \notin D$. In protocol discovery, this determination is made based on feedback from interaction with the network, primarily through communication attempts with various protocols. Until negative feedback arrives about a protocol in p_i, $p_i \in D$ still may be true. In general, $p_i \notin D$ can only be known when feedback arrives from some protocol indicating that p_i, or some protocol in p_i, is not supported on D. Because of intermittent failures and delays, it is not possible to assume $p_i \notin D$ based only on the failure of an attempt to use p_i. On the other hand, positive feedback about $p_i \in D$ may come as a result of successful interaction with all of the protocols in the path p_i or from another protocol indicating that p_i is supported on D.

In protocol discovery it is not necessary to determine the entire set $p_i \in D$. All that is needed is to find one common path between D and S that uses the same protocols and will therefore provide sufficient communication. The discovery process performs a bottom-up search of the protocol graph to find a matching protocol path. The discovery system starts with a low level protocol, like the CLNP network layer, and tries to establish whether it exists on the destination. Once positive feedback arrives from the network layer, the next step is to determine which transport layer protocols are available through this network layer. Eventually, when communication is established with some application that provides the service requested, the discovery process can stop and normal communication can proceed. In this case we say that a complete, matching protocol path has been found. It is possible that a partially successful communication attempt will not lead to complete success. For instance, even after identifying a network and transport layer, it may still be necessary to backtrack and find another network layer if the first one does not lead to an application providing the desired service.

We have identified the following two basic approaches to supporting discovery in multiprotocol systems.

- **User Discovery:** In this approach, the user performs discovery by attempting to use different protocol paths. This is the approach carried out in the previous example using FTP and FTAM. To support this approach it is necessary to enhance the current protocol systems to provide explicit failure indications. In the above scenario, a message such as *CLNP supported, TP4 not supported* rather than the simple *Connection refused* would greatly assist the user in determining which protocol path to attempt next.
- **Automated Discovery System:** This approach involves implementing a protocol subsystem that performs protocol discovery automatically as part of the normal protocol operations. Such a system could receive a communication request from the user such as, "Transfer files from host A to host B", and then discover which protocols to use to perform the task.

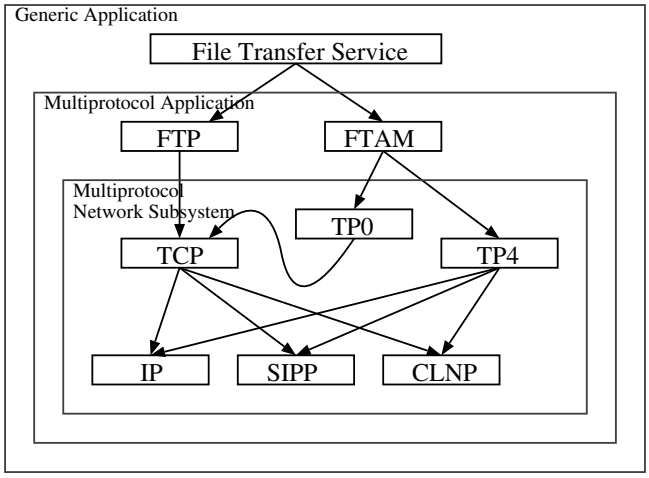

Figure 4: Automated Discovery Architecture Scopes

In the current work, we focus on the second approach. A goal of this research is to develop the necessary components for a discovery system that can take a user's request and return a "connected" communication session. This system will operate without the user being aware of the protocols used, different network address formats, or failures during protocol attempts. In this system, feedback is given to the user only after the communication is successfully established or all possible combinations are exhausted.

3.3 Automated Discovery Architectures

Here we describe three different architectures for developing an automated discovery system. Each architecture has its own virtues and limitations. As we will see, the main distinction among the architectures is the scope of the discovery system in terms of the protocols included. This difference in scope is depicted in Figure 4.

Generic Application: The first approach to automated discovery is to perform protocol discovery as part of a generic user application that provides a common service. The generic File Transfer Service of Figure 4 presents the user with a consistent interface, regardless of the actual protocols or applications used [17]. This discovery approach incorporates the entire protocol graph, including the applications, into the discovery system. The main advantage of this approach is that it allows the user to communicate with a wide range of currently-installed systems, including those supporting a variety of applications.

A problem with this approach is that it is difficult to hide the actual applications and some limitations will always arise when developing the generic interface. For instance, the FTAM application includes several features not found in FTP. These

features will need to be either emulated for FTP communications or not provided at all.

Multiprotocol Application: The second approach to performing automated protocol discovery is to develop multiprotocol versions of applications that perform discovery themselves. For example, a multiprotocol FTP implementation could support discovery by including calls to several different protocols[4]. This approach provides the user with a familiar user interface and functionality and while hiding many of the details of protocol discovery. The user will still need to perform part of the discovery process by selecting the application that is supported on the remote host.

One disadvantage of this approach is that it only provides communication with systems that support some version of this multiprotocol application. In the short term, this application will primarily be supported only over its native protocols (e.g., FTP over TCP/IP). This will limit the connectivity attainable by the discovery system. Another drawback is that this approach may require extensive modification of the application to support new protocols and address formats.

Multiprotocol Network Subsystem: The effort needed to modify the application to support multiple protocols can be alleviated by performing discovery below the application/protocol interface. This is the idea behind our third approach where a multiprotocol network subsystem is used to provide multiprotocol support through standard programming interfaces. The network subsystem is the portion of a host operating system that supports protocol implementations. Two popular examples are the System V Streams [2] and BSD UNIX Socket [11] environments. Implementing the discovery algorithm as part of the network subsystem enables current applications to run over multiple protocols with little or no modification to the actual application. The degree of connectivity provided is essentially the same as in the multiprotocol application approach.

The main drawback of this approach is that it requires extensive modification of the network subsystem. Also, as with the generic application, the use of a generic interface to many protocols will usually result in a compromise of functionality for some of the protocols. For instance, TCP provides a graceful disconnect while the OSI TP4 does not. Another issue is that since the discovery process is done automatically, the application programmer loses some control over the actual protocols used as well as the discovery process.

[4]Some of the changes needed to provide this support in FTP have already been proposed [15].

3.4 Issues With Discovery Algorithms

Each of the discovery architectures proposed in the previous section incorporates some form of algorithm or set of rules that directs the operation of the discovery process. The design of these algorithms is affected by the choice of discovery architecture as well as other factors. In this section we present several issues in developing discovery algorithms and discuss how these issues relate to the different architectures.

- *Where to start:* The first interesting question in designing a discovery algorithm is, "Which protocol should be tried first?" This choice can be completely random or it can be based on some *a priori* knowledge of protocols that have a better than average possibility of success. For instance, if a multiprotocol system is connected to a network where most of the other hosts support one specific protocol family, then those protocols should probably be tried first. Additionally, a cache of recently contacted hosts could be used to store information about the protocols successfully used with these hosts.

  Another good choice for a first try is a protocol that provides particularly good feedback when failures occur. In this case, even if the protocol does not establish the desired communication, it should provide good insight into which protocols to try next.

- *Interpreting failure:* The most important aspect of the discovery process is the use of failed communication attempts to learn as much as possible about the remote configuration. The information learned is then used to guide the selection of the next protocol to be attempted. The extent to which this is possible depends on the type of feedback provided by a protocol when it fails. In Section 3.1 we presented an example where more detailed feedback would be useful for user discovery. In the next section we give more detail on the type of feedback needed for automated discovery systems.

- *When to give up:* Many failed attempts will result in no feedback at all. In these cases it is difficult to distinguish a situation of temporary network congestion or failure from a case where the protocols being used are not supported on the remote system. The first decision of this sort is to decide when to stop trying one protocol and move on to another. Additionally, the discovery algorithm may be designed to start over after unsuccessfully trying all the protocol combinations and try them again. In this case the system must decide when to abandon this discovery cycle.

- *Datagram vs. Connection Service:* Discovery fits most naturally as part of the connection establishment process. This is because of the fact that positive feedback of communication success is provided as part of connection establishment. Datagram service is often unreliable and a “no feedback” situation could mean that the communication was a success. In this case, higher level

indication from the application or user is necessary to provide enough context to indicate when a communication attempt was successful. The generic application and multiprotocol application architectures are more likely to be able to incorporate this application level context into the discovery process.

- *Access to Protocol Operation:* The feedback mechanism supported by a protocol is only useful for protocol discovery if the discovery system has access to it during each communication attempt. In many cases, protocol implementations do not make the complete feedback information available to the protocol users. It may also be useful for the discovery system to have access into the operation of the protocol. One example of this is a system that uses TCP as one of the possible transport protocols. While the TCP handshake is taking place, some feedback has been received but the upper layer has not yet heard indication of a connection. If the discovery system is about to give up on this attempt, it could check on the state of the TCP protocol and detect that enough progress has been made to warrant waiting further.

  In order to provide this type of access to protocol operations it is necessary to "open up" the details of the actual protocol implementations. This will be difficult for the generic application and multiprotocol application architectures since they will likely be implemented as user level programs. Such programs will have protection problems in getting to the detailed protocol information in popular systems like UNIX where the protocols are commonly implemented in a privileged kernel space.

- *Parallel attempts:* The discovery algorithm can be designed to try protocols one at a time or it could try several protocols in parallel. In the parallel case, all the possible protocol combinations would be tried at the same time and the first one to successfully communicate would be handed off to the user. The parallel approach might be practical when parallel hardware is available or when there is a long propagation delay between the two systems and trying several protocols is faster than waiting for feedback.

  One problem with this approach is that it may result in the creation of several successfully communicating sessions, all but one of which would need to be gracefully terminated without being used.

3.5 Feedback

The feedback provided by protocols regarding communication failures is important enough to the discovery process to warrant further discussion. As we described in Section 3, the only way a discovery system can determine for sure whether protocol path $p_i \in D$ or $p_i \notin D$ is to obtain some definite feedback indicating which one is true.

The most effective way to obtain positive feedback about the existence of p_i is to get it directly from the protocols that constitute p_i. This feedback could be in the form of a direct acknowledgment of transmitted data or it could be any other data received from the destination that uses the protocols in p_i. This second approach is necessary when doing discovery with unreliable datagram protocols that do not send back acknowledgments. It is also possible to get feedback from other protocols, which are not part of p_i, indicating that p_i is supported on D. While such feedback would strongly suggest $p_i \in D$, the only sure way to guarantee that p_i is truly available for communication is to hear it directly from the protocols in p_i.

Unlike positive feedback, negative feedback indicating $p_i \notin D$ will, in general, need to be sent by some protocol other than the one missing from D. Most of the time, useful negative feedback will come from some lower layer protocol in p_i that carries the traffic of the missing protocol. For instance, a network layer protocol might return an error indicating that the desired transport layer protocol was not found. The only example we have encountered where negative feedback might come from the protocol itself is the case of network protocols that send feedback from an intermediate router (e.g., $I1$ in Figure 3) indicating that D is not reachable using this network protocol.

Clearly it is impractical for D to provide complete feedback for every protocol message sent by S; in the case of an unreliable connection-less protocol, this would conflict with the design intention of the protocol. One option that may be useful for such protocols is to provide a variable feedback mechanism that allows the sender to request positive feedback on certain data. This feedback could be turned on by S for the first several datagrams while discovery is taking place. After the protocols are determined and communication succeeds, the feedback could be turned off to provide the efficiency normally desired for such protocols.

CLNP provides a two level feedback mechanism that allows the source to specify whether or not negative feedback should be returned. Feedback is requested by setting the error report (ER) flag in the header of each packet for which feedback is desired. We propose that future protocols include a three level feedback mechanism. The feedback level is indicated by the value of a 2-bit field in the protocol header. This field tells the end and intermediate systems what type of feedback should be sent to S about the processing of this packet. The highest feedback level indicates both positive and negative feedback. This is the level used during protocol discovery. It allows both end and intermediate systems to provide an immediate indication of the degree of success achieved with this protocol. After determining which protocols to use, the source sets the feedback field of all subsequent outgoing packets to request only negative feedback. This is the normal case for most feedback mechanisms today. The third feedback level indicates that no feedback should be provided to S regarding this packet. This option is useful for any application where feedback is unnecessary or could overwhelm the source.

4 Discovery Implementation

To demonstrate the feasibility of the protocol discovery concept we undertook an implementation incorporating two of the three network protocols from Figure 1: IP and CLNP[5]. This implementation was done in Sun OS 4.1.3 for the Sun Sparc architecture. This work involved the addition of the CLNP protocol[6] as well as the development of the discovery system we describe.

We describe our experience and insights gained from this implementation work in this section. We pursued both the multiprotocol application and multiprotocol network subsystem architectural approaches (See Section 3.3). As part of our implementation, we developed the discovery algorithm presented in Figure 5. This algorithm performs discovery of protocols that provide a connection-oriented service. We developed this algorithm after carefully analyzing the feedback provided by these protocols for various compatibility problems. Before describing the algorithm we summarize our findings regarding this feedback.

4.1 Feedback Analysis

With IP, network layer feedback is provided by the Internet Control Message Protocol (ICMP) [16]. ICMP is an unusual protocol in that it is both an integral part of IP and a user of IP, using IP to transfer its messages. ICMP has several different message types, most of which are used to provide feedback during communication. These messages are generated by a network host when it encounters an error while processing an IP datagram.

Table 2 gives the ICMP feedback that would be generated for several possible protocol compatibility problems. The most difficult problem to recognize is when there is no compatible network layer at $I1$, the next hop from the source. When this occurs, there is no direct feedback and therefore a failure can only be inferred by a timeout in a higher layer protocol. For the other compatibility problems listed in Table 2 there are unique feedback messages in ICMP to indicate the problem. These messages can be used by a multiprotocol system to determine where an incompatibility occurs. The feedback presently proposed for SIPP is a straightforward extension of the current ICMP [9]. In the current proposal, there are few differences between the ICMP feedback messages for IP and SIPP.

Unlike IP and SIPP, CLNP includes a feedback mechanism as part of the network protocol definition. CLNP supports a number of *Error Report (ER) PDUs* that

[5]The SIPP protocol is still evolving. We have been closely following the standardization effort and will be incorporating SIPP into our implementation as the specification matures.

[6]For the CLNP portion we made significant use of code from the NetBSD software release. Additionally, we are indebted to Francis Dupont for the contributions made to support TUBA.

Compatibility Problem	ICMP Feedback	Generated by
No IP at *I1*	None, timeout	*S*
No IP at *I2*	Net Unreachable	*I1*
No IP at *D*	Host Unreachable	*I2*
No matching Transport at *D*	Protocol Unreachable	*D*
No matching Application at *D*	Port Unreachable	*D*
IP option mismatch at *I1,I2,D*	Parameter Problem	*I1,I2,D*
Time Exceeded at *I1,I2*	Time Exceeded in Transit	*I1,I2*
Time Exceeded at *D*	Time Exceeded in Reassembly	*D*

Table 2: Protocol Problem Feedback for IP

provide feedback during the operation of the network layer protocol. Most of the ER messages are designed to provide specific feedback for the operation of CLNP itself. The ER PDUs returned for various protocol compatibility problems are given in Table 3. Two important messages that are not provided by the CLNP ER PDUs are the Protocol and Port Unreachable messages found in ICMP. As we will discuss in Section 4, these messages are particularly useful in multiprotocol networks where there may be several different transport and application layer protocols. Like ICMP, there is no means for CLNP to automatically detect that the protocol is not supported on the next hop *I1*. A timeout mechanism is also required in this case to detect such a failure.

Compatibility Problem	CLNP Feedback	Generated by
No CLNP at *I1*	None, timeout	*S*
No CLNP at *I2*	Destination Unreachable	*I1*
No CLNP at *D*	Destination Unknown	*I2*
No matching Transport at *D*	None	*D*
No matching Application at *D*	None	*D*
CLNP option mismatch at *I1,I2,D*	Unsupported Option	*I1,I2,D*
Time Exceeded at *I1,I2*	Lifetime Expired in Transit	*I1,I2*
Time Exceeded at *D*	Reassembly Lifetime Expired	*D*

Table 3: Protocol Problem Feedback for CLNP

4.2 A Discovery Algorithm

The discovery algorithm we developed starts at the upper left of Figure 5 with the TCP/IP protocols. We chose this as our starting point since a large percentage of the systems on our network support these protocols. If the attempt succeeds then the connection is established and communication proceeds. If the attempt fails, this algorithm takes one of three different paths depending on the feedback given.

If an ICMP *Protocol Unreachable* is received then we know that IP is supported on the remote system and it is probably best to try a different upper layer protocol. If a TCP *Reset* message is received then we know that both IP and TCP are supported but the requested application was inaccessible through TCP. The system could still support this application over TCP using the RFC-1006 mechanism for providing OSI applications over TCP. If TCP is found, the RFC-1006 option is attempted next. If TCP is not found but IP is, then the TP4 protocol is attempted with IP. If neither of these feedback messages are received, the system assumes that IP is not supported on the remote system and goes on to try the TCP/SIPP combination.

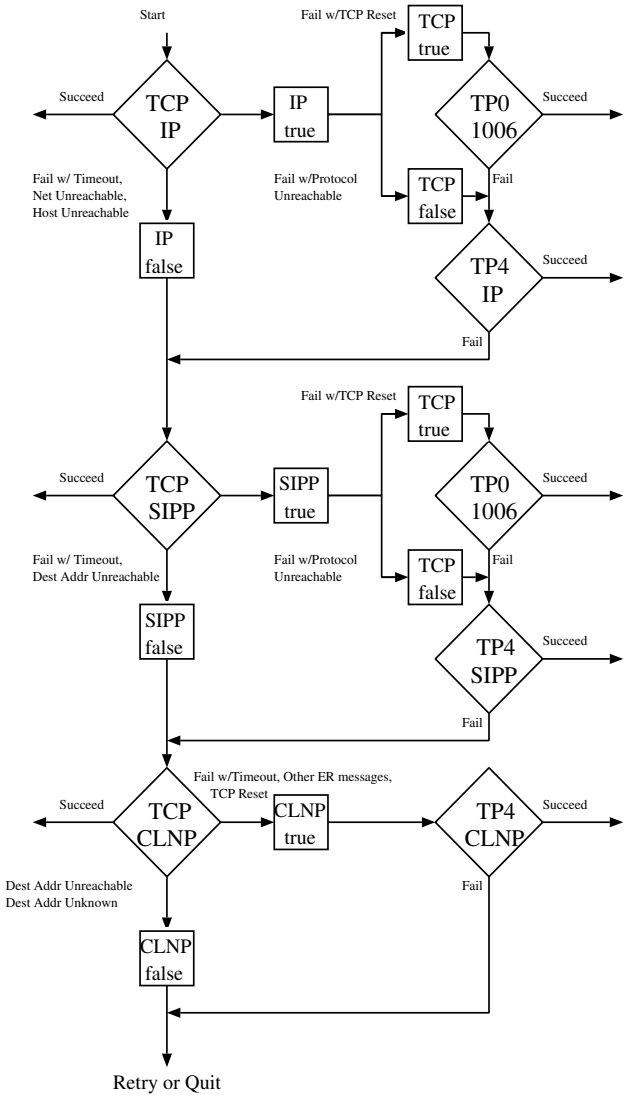

Figure 5: Protocol Discovery Algorithm

The SIPP portion of the algorithm proceeds much like the IP portion since the ICMP feedback is similar for the two protocols. In our implementation, we do not yet support the SIPP protocol so this portion of the algorithm is not used. The CLNP algorithm does not include the two options for higher level protocol problems since

it does not include specific feedback about unreachables[7]. For CLNP, if a packet arrives for an unsupported upper layer protocol the packet is simply discarded. This means that the case of no feedback from a CLNP attempt could still mean that CLNP is supported on the system. When TP4 is found on the remote system but the application is not available, TP4 will return a *Disconnect Request* TPDU. In the algorithm presented here, this feedback does not affect the order of protocol attempts.

Multiprotocol Application: In our implementation of the multiprotocol application architecture we developed a version of the discovery algorithm as part of a simple data transfer application. For each protocol combination attempted by the algorithm, the application creates a new socket using the appropriate protocols. The standard socket interface does not require the level of feedback required by our discovery algorithm. In order to provide feedback regarding the actual failures we added several new error codes (i.e., errno values) that indicate the ICMP return codes and receipt of a TCP Reset.

Multiprotocol Network Subsystem: To implement the multiprotocol network subsystem, we incorporated the discovery algorithm into the BSD socket architecture. In this implementation we introduced the novel concept of a *multiprotocol family*. This new protocol family is denoted as PF_MULTI. To use the multiprotocol system, a programmer creates a socket with this protocol family and the protocol service type required (e.g., datagram or stream). When the socket is created it is still not known exactly which protocols will be used to implement this socket. The discovery algorithm is invoked when the user attempts to establish a connection via the *connect()* system call. After discovery finds the protocols to be used, the protocol specific values are filled in to the socket structure. For datagram service, protocol discovery is performed at the time of the first data send (e.g., with the *sendto()* system call).

It is important to note that our multiprotocol subsystem version of automated discovery is implemented within the UNIX kernel and has access to the entire set of protocol implementations and the feedback they provide. When implementing an automated discovery system in other architectures where protocols are not part of the same privileged address space (e.g., Mach [1]), it will be important to provide access to the protocol feedback systems.

Addressing: One interesting challenge with this architecture is how to specify the appropriate address information for each of the protocols that will be attempted. In current systems, the application creates an address structure of the appropriate

[7]The TUBA proposal [3] recommends the addition of these as two new ER types.

type (e.g., AF_INET) and passes it in as an argument to the *connect()* system call. With our system, it is necessary to have addresses for each of the several different protocols that will be attempted. We provide these in the *connect()* call as a linked list of address structures (sockaddr). We felt this to be a better approach than the alternative of having the PF_MULTI domain code obtain appropriate addresses for all the protocols to be tried.

Both of our implementations provide protocol discovery that enables the user of an application on a multiprotocol system to communicate with hosts supporting any of five different protocol combinations. The protocol subsystem approach provides this support without requiring the application programmer to implement the discovery algorithm.

5 Concluding Remarks

One approach to providing interoperability in a contemporary network is to develop systems that support several different protocols. Protocol discovery is an effective mechanism for handling the problem of determining which of several protocols to use. In this work we describe three main architectures for developing protocol systems that automatically perform discovery. These architectures offer varying levels of interoperability with other network systems. While the generic application offers the user seamless connectivity to the widest range of systems, this approach has the highest development cost since much of the implementation can only be used for the specific application it was developed for. The multiprotocol subsystem approach allows a single discovery implementation to support several different applications.

While it is possible to implement protocol discovery with current protocols and protocol implementations, the discovery process can be greatly enhanced by some simple additions. First, protocols should provide feedback indicating whether or not they succeed in reaching the destination system and whether or not the next higher level protocol was found. We proposed a simple variable feedback mechanism that provides this feedback. Protocol designers should consider including this type of feedback in future protocol standards. Second, the protocol implementations should be designed to make this feedback available to the users of the protocol.

The discovery algorithm we presented here was designed after analyzing the feedback provided by the protocols in use, studying the implementation options in our development environment, and using empirical evidence to decide which protocols were the more likely to succeed after each failure. Our future work will include the continued study of discovery algorithms for different protocols and the exploration of ways to simplify the design of discovery algorithms. We will also

be pursuing the implementation of SIPP in our current architecture once the SIPP protocol standard solidifies.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for Unix development. In *Proceedings Summer Usenix*, July 1986.

[2] AT&T. *STREAMS Programmer's Guide*, 1988. Unix System V.

[3] R. W. Callon. TCP and UDP with bigger addresses (TUBA), a simple proposal for internet addressing and routing. *RFC 1347*, June 1992.

[4] R. J. Clark, M. H. Ammar, and K. L. Calvert. Multi-protocol architectures as a paradigm for achieving inter-operability. In *Proceedings of IEEE INFOCOM*, April 1993.

[5] R. J. Clark, K. L. Calvert, and M. H. Ammar. On the use of directory services to support multiprotocol interoperability. In *Proceedings of IEEE INFOCOM*, June 1994.

[6] D. E. Comer and J. C. Lin. Probing TCP implementations. In *Summer USENIX*, pages 245–255, June 1994.

[7] S. Deering. Simple internet protocol plus (SIPP) specification. *Internet Draft*, July 1994.

[8] J. R. Falcone. A programmable interface language for heterogeneous distributed systems. *ACM Transactions on Computer Systems*, 5(4):330–351, November 1987.

[9] R. Govindan and S. Deering. ICMP and IGMP for the simple internet protocol plus (SIPP). *Internet Draft*, March 1994.

[10] P. Janson, R. Molva, and S. Zatti. Architectural directions for opening IBM networks: The case of OSI. *IBM Systems Journal*, 31(2):313–335, 1992.

[11] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.3 BSD UNIX Operating System*. Addison-Wesley, 1989.

[12] B. Leiner and Y. Rekhter. The multiprotocol internet. *RFC1560*, December 1993.

[13] B. Meandzija. Integration through meta-communication. In *Proceedings of IEEE INFOCOM*, pages 702–709, June 1990.

[14] D. M. Ogle, K. M. Tracey, R. A. Floyd, and G. Bollella. Dynamically selecting protocols for socket applications. *IEEE Network*, 7(3):48–57, May 1993.

[15] D. Piscitello. FTP operation over big address records FOOBAR. *RFC1639*, June 1994.

[16] J. B. Postel. Internet control message protocol. *RFC 792*, September 1981.

[17] M. T. Rose. *The Open Book*. Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[18] M. T. Rose and D. E. Cass. ISO Transport Services on top of the TCP. *RFC 1006*, May 1987.

[19] C. Tschudin. Flexible protocol stacks. In *Computer Communication Review*, pages 197–205. ACM Press, September 1991.

[20] M. K. Vernon, E. D. Lazowska, and S. D. Personick. *R&D for the NII: Technical Challenges*. Interuniversity Communications Council, Inc. (EDUCOM), 1994.