An Extensible End-to-End Protocol and Framework

K. L. Calvert
R. H. Kravets
R. D. Krupczak
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, USA
{calvert,robink,rdk}@cc.gatech.edu

Abstract

We describe a framework for composing end-to-end protocol functions. The framework comprises: a generic model of protocol processing; a metaheader protocol supporting per-packet configuration of protocol function and efficient demultiplexing of incoming data units; and an extensible set of modular protocol functions. This paper describes the pieces of the framework and motivates some of the design decisions.

1 Introduction and Motivations

As requirements for high-speed networking continue to evolve, it is difficult for the current protocols and protocol architectures to keep up. On the one hand, no one can predict what the next generation of applications will demand from the network. This suggests the need for flexibility and extendibility. On the other hand, it has been shown that the traditional mechanism for composing protocol functions —i.e., layering— can be a performance impediment. This suggests the need for a new mechanism, supporting flexible combinations of functionality *without* layering.

This paper describes τ (**t**ransport **a**nd **u**p), a protocol that provides such a composition mechanism, allowing a very general assortment of end-to-end protocol functions to be combined in support of a wide variety of applications. Examples of such functions include error detection, encryption, flow control, connection management, bandwidth management, various levels of reliability, and so on. Because the variety of applications we envision includes those for which performance (especially bandwidth) is important, the composition mechanism of τ is friendly to performance-oriented implementation techniques, in particular to the integration of data-handling functions [AP93, CT90] and parallel processing [ZT93, NYKT94, Bjö94]. Thus, in addition to being a protocol that defines the form and meaning of "bits on the wire", τ is also a *framework*, which defines the possible interactions between protocol functions. As such, τ is designed to accommodate functions that reside between two natural layer boundaries: above the (inter)network layer and below the application.

In terms of Feldmeier's taxonomy of architectural concepts for high performance [Fel93], τ supports reduced processing requirements through separation of control and data functions, configurability, and elimination of duplicated functionality. It also supports increased processing availability, via parallel implementation and data-handling integration.

The remainder of this paper is organized as follows. The next section discusses related work, and ideas that have influenced the development of τ. Section 3 describes τ's general design and philosophy. Section 4 presents some details of the metaheader protocol and the generic interface. Section 5 offers some concluding remarks.

2 Related Work

This section presents the main ideas that have influenced the design of τ.

Several researchers have identified *layered multiplexing* as an impediment to protocol performance [Fel90, Ten89]. In addition to detrimental effects on flow and congestion control, layered multiplexing forces the state information relevant to an incoming data unit to be retrieved sequentially, one protocol at a time. As Feldmeier has shown, however, for most reasonable models of state storage, it is faster to retrieve state information all at once [Fel90]. Unfortunately, with traditional layering, this is difficult, especially when one layer may transform the header of another.

Clark and Tennenhouse [CT90] identified poorly-structured *layered* implementations as a performance bottleneck because they may require separate passes over the data for each layer, an expensive approach in modern computer architectures. They proposed *integrated layer processing*, in which all data-touching operations are performed in a single pass over the data. Such an implementation approach does not preclude layering as a way of *logically* structuring protocols, but it does require that all headers attached to a piece of data be (sequentially) processed before the data itself can be processed. Again, when the header of a higher layer is treated as data by a lower layer that performs encryption or other transformations, this may not be possible. Moreover, this form of integration hinders certain other optimizations, such as dynamic inclusion and exclusion of protocol functions [OP92, Zit91]. To get around this, it has been proposed that different functions be "next to, rather than on top of" each other [CT90].

Another key observation [CT90] is that whenever data "comes to rest" inside the protocol stack —for re-sequencing, fragment reassembly, or whatever— this "lost time" can never be made up if the protocol stack is the bottleneck between the application and the network. This leads to the idea of *application layer framing*: the communication subsystem should respect the data unit boundaries indicated by the application, and sufficient information should be included in each (application) data unit to enable the receiving application to deal with it, regardless of its order with respect to other data units. By the same token, the protocol stack should deliver data units as they arrive, rather than enforcing order on applications that may not need it.

Protocol configurability, the idea that an application should be able to select from a "menu" exactly those protocol functions[1] it requires in order to reduce overhead and increase performance, has been addressed in a number of recent projects [Haa91, Zit91, PPVW93, PS91, LKAS93a, FRS93]. Some of these have attempted parallel implementations to improve performance [LKAS93b, Bjö94, Zit91]. To our knowledge, all of these approaches focus on a closed set of protocol functions, in the context of traditional layered encapsulation. The distinctive feature of τ is that it abandons the traditional *layered encapsulation* approach to protocol composition, in order to support performance-enhancing techniques like those just discussed. The basic idea is to replace layering with an *explicit* mechanism (a meta-protocol) for composing functions [Cal93], while also providing a single, uniform multiplexing function for all upper-layer protocols.

[1] We use the term "protocol function" in a manner similar to Tantawy and Zitterbart [ZT93], to denote an "atomic" unit of functionality.

3 Design Overview

The primary design objective of τ is to provide two mechanisms (*protocol function composition* and *multiplexing*) in a manner that supports various performance-enhancing techniques, while preserving modularity in some form. The idea is that these mechanisms should work with an extensible set of *policies*, in order to support a wide variety of applications, including those whose requirements are not yet fully understood. Other specific design goals for τ as an upper-layer protocol include:

- Support for both *integrated data-handling* and *dynamic inclusion-exclusion* of protocol functions, through explicit early identification of the functions applied to a data unit.
- Support for enhanced *reusability* of protocol function implementations, through a generic protocol interface by which protocol functions are accessed.
- Support for *multicast* applications.
- Support for various forms of *encryption*.

The τ framework comprises three parts:

1. A *generic model of protocol processing*, in which the protocol functions are separated from the details of the "glue" that binds them together.
2. A *metaheader protocol*, which provides the basic mechanism for implementing the primary functions of τ, namely multiplexing and non-layered flexible protocol function composition.
3. A set of modular *protocol functions*, structured according to the generic model.

We discuss each of these three parts in turn.

3.1 A Generic Model of Protocol Processing

Because protocol functions are not layered in τ, they do not attach their headers directly to outgoing data units, nor extract them from incoming data units; instead, this is handled by the τ infrastructure. The generic protocol model defines the interface between the τ infrastructure and each protocol function. The requirement that this interface be specified represents an opportunity to reduce the costs of porting protocol implementations by isolating, as far as possible, the specification of a protocol's function from the "glue" used to combine it with other protocols. This is a key design motivation of τ.

A logical block diagram of a τ implementation is shown in Figure 1. Each *protocol function module* is viewed as a (passive) transducer, which is given inputs (state information, control parameters, incoming header, and possibly data) and produces outputs (updated state information, control parameters, outgoing header, and processed data). The architectural "glue" is provided by the τ demux-and-dispatch function. It selects and coordinates between the protocol functions, providing them with inputs based upon external events, and collecting and passing on to the external environment (i.e. the user, the network, auxiliary functions such as timeout and buffer management) their outputs. Logically, each protocol function module operates independently and concurrently on any given message—except for data-handling functions, which process the data in a message according to the order indicated by the τ

metaheader, and which may be *integrated*, so that they do so in one pass over the data. (This is indicated in the figure for three of the data-handling functions.) In the implementation, however, protocol functions may be invoked sequentially by the demux-and-dispatch function.

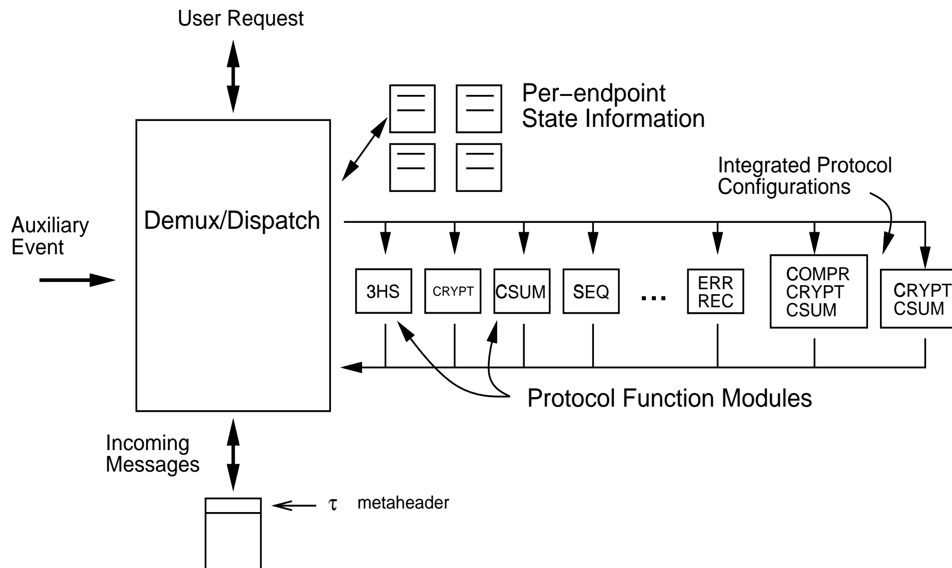

Figure 1: Architecture of τ

The interface is defined in terms of a set of *entry points* corresponding to various external stimuli: incoming data unit, user request, timer expiry, etc. In order to exploit the global information available to the τ infrastructure and pass it along to the protocol functions themselves, the interface to each protocol function consists of a relatively large number of relatively special-purpose entry points. This allows the protocol function to base its behavior on the specific situation indicated by the entry point, without having to first decide what the situation is. For example, there are two `send` entry points, one a "fast path" send, used when desired functionality has been preselected and throughput is paramount, and the other a "general path" send, which is used for functions that may not have been configured in advance.

The generic interface to protocol functions supports the principles of Application Layer Framing [CT90] by assuming that data does not "stop" within τ, either on the way in from the network or on the way out from the user. That is, the τ generic protocol model stipulates that functions be composable in such a way that data *need not* be buffered in τ. Thus, the user must provide buffer space *before* any incoming data units will be processed. Similarly, user requests to send data will block, and do not transfer any data until the *flow control* function (if one is present) says it is OK to do so. Moreover, data-handling protocol functions must be defined in such a way that they can be *integrated*, i.e. so that they can operate on the data in "one pass".

Note that the functions need not actually *be* integrated in any particular implementation; the constraint requires that functions (at least those that adhere to the generic protocol model) must be designed to be "integrable". Indeed, our position is that integration of data-handling functions is of little benefit unless the functions are designed specifically for integration. Integration of a checksum function with DES encryption, for example, offers little performance improvement, because the huge expense of DES overwhelms all other considerations. The extensibility of τ supports introduction of new functions, designed to be integrated with others, as the need arises.

Because control functions are logically processed in parallel in our generic model, some means is required to allow one function —a data integrity check (checksum), for example— to "abort" or "override" the processing of others. The need for this capability has been noted by others [Haa91, AP93]. In τ, protocol functions update state information (and the user is notified of incoming data) according to a two-step process, where every function must agree on the acceptability of the received data unit, and only then, when all are agreed, do they all commit their state updates.

Certain other interactions among protocol functions fall outside the generic model; these are discussed later in the paper.

3.2 Metaheader Protocol

The τ metaheader protocol implements the multiplexing and composition mechanisms. As such, it contains two kinds of information:

- Information the receiving τ can use to efficiently retrieve the relevant state information for the incoming data unit (i.e. *demultiplex* it to the right place), and
- Information indicating which set of protocol functions need to be applied to the incoming data unit, and, in the case of data-touching functions, in what order they should be applied.

Every τ metaheader includes identifiers from two number spaces:

- **Application Identifiers** tell what application the data unit is supporting at the next higher level. IDs from this space can encode information about the behavioral attributes of the application, e.g. multicast, involves encryption, transaction-oriented, etc.
- **Protocol Function Identifiers** identify the protocol functions applied to a data unit. The identifier assigned to a protocol function indicates whether it is a data-touching function, and whether it is symmetric or asymmetric (and in the latter case, distinguishes between the two ends).

Identifier spaces in τ are capacious enough to encode any additional information that might be useful. They are divided into "well-known" and "local" subspaces (as with Internet Protocol transport address spaces); the idea is that the former have globally-agreed significance, while the latter may be assigned locally. Both have portions reserved for encoding existing Internet identifiers (e.g., IP protocol numbers may be used as protocol function identifiers).

The main design issue for the metaheader protocol is how much τ needs to "know" about *specific* protocol functions. For instance, is it necessary for the τ infrastructure to recognize that an incoming data unit contains a *connection management* header? Or is it possible to treat connection management within the generic model? Our approach has been to keep specific functions from being reflected in the τ metaheader and keep the mechanisms as generic as possible in order to support the widest variety of policies. There are, however certain functions that the τ infrastructure needs to deal with in a special way; some of these are considered in the next section.

3.3 Protocol Functions and τ

We envision communication services implemented by composing atomic single-function protocols from a "menu of functionality", as have others [OP92, ZT93, Haa91, PPVW93, PS93, SBS93]. For example, a service for a transaction-oriented application could be implemented with a soft-state connection management function, a sequence numbering function, and two different reliability functions (one for request retransmission and one for response error detection and retransmission).

As noted earlier, our generic model of protocol processing assumes that user data does not stop within the τ subsystem. This permits interactions between protocols to be constrained to occur along a rather narrow interface. However, it also imposes certain restrictions on what and how protocol functions may be composed. For example, a *fragmentation/reassembly* function cannot be used under this restriction, because in general fragments have to wait for other fragments before the data unit can be processed.

We have identified several protocol functions that require "special-purpose interfaces" within τ, including:

- *Fragmentation and reassembly*: because this function transforms a single unit into multiple units (and vice versa), it cannot be "integrated" or performed in parallel with other functions that operate on a single data unit.
- *Header Encryption*: τ must know whether to decrypt header information before dispatching it to individual protocol functions.
- *Flow/congestion control*: τ must check whether flow control (or congestion control) allows transmission of a data unit before beginning processing of a send request, to avoid buffering data.
- *Reply-to/return address*: it is τ's responsibility to place the multiplexing information in the metaheader of any message sent in response to another message.

The τ metaheader includes an explicit mechanism for only one of these: header encryption. Others are handled via special "intra-τ" interfaces. In the cases of fragmentation, the function can be used within τ, but it may not be composed with any other function. To use either of these functions with another, τ is used *recursively*, e.g. τ-with-fragmentation treats τ-with-other-functions as its *user*, and one τ metaheader (with other headers) is carried as user data inside another τ data unit. The approach is described in more detail in Section 4.2.2; one particular form of encryption has a similar restriction. The performance penalty associated with this approach —due to layered multiplexing and forced non-integrated data handling— nicely characterizes the overall design philosophy of τ: removal from the "fast path" is an acceptable price for generality.

4 Design Details

4.1 The τ Metaheader Protocol

The general format of a τ data unit is shown in Figure 2.

The τ protocol supports dynamic modification of protocol configurations by having each data unit be self-describing (via its metaheader). The protocol allows for different levels of protocol configuration. The first level includes configurations that are globally well-known; these are generally associated with a

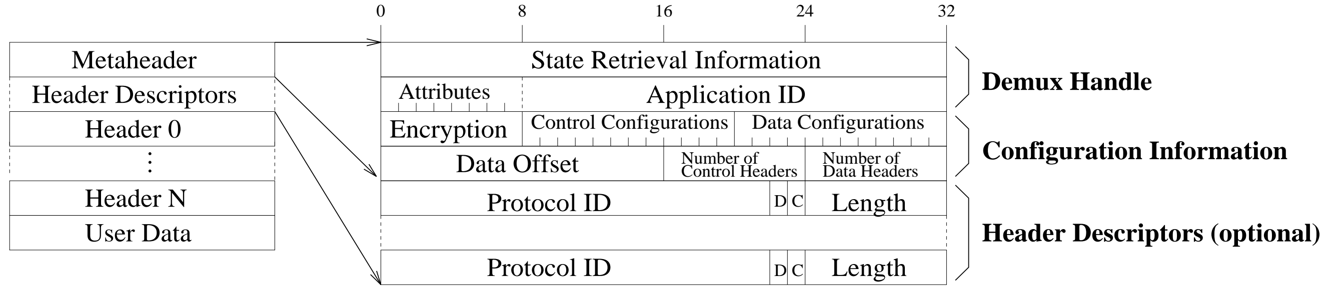

Figure 2: τ Metaheader Format

specific application ID. For example, a client-server application might define three header configurations, to be selected on the basis of the amount of data the client expects to receive in response to its request. The second level of configuration is defined at connection start time. These are configurations that are agreed upon dynamically by the communicating endpoints, possibly after some negotiation. These first two levels of configuration make use of the "Data configuration" and "Control configuration" fields in the metaheader to identify which agreed-upon configuration of protocol headers is present in the message. When this information is present (indicated by a nonzero value in these fields), Header Descriptors need not be present.

The third level of configuration is explicitly defined per-message. Any endpoint may use a specific configuration for a specific message. In this case, the "Data configuration" and "Control configuration" fields are zero, the "Number of control headers" and "Number of data headers" fields are nonzero, and the metaheader is followed by one or more Header Descriptors. Any necessary information about that configuration is conveyed in the header descriptors. Thus, the application has the choice of trading off the speed and simplicity of using well-known configurations with the flexibility that may be needed on a per-connection or per-message basis.

The metaheader has two main parts —the *demux handle* and the *configuration information*— corresponding to the two functions it implements. The metaheader (and the header descriptors, if present) is followed by the headers themselves, which carry the protocol control information of the various protocols functions. The

4.1.1 The Demux Handle

The **Demux Handle** is made up of the state retrieval information and the application information. Together, these two pieces of information determine how τ handles the message.

- The **state retrieval information** (SRI) is used by the recipient to locate any state information relevant to this data unit. The SRI is similar in function to the *reference number* used in the ISO transport protocol [fS86]. It is chosen by the recipient and communicated to the sender(s) either at connection setup time or through some other channel. An SRI value of 0 indicates that the data unit is not associated with any existing state information. In keeping with separation of mechanism and policy, the structure of the SRI is not specified by τ. One possible approach is to use a portion of the 32 bits as a hash key, and the rest as a "magic cookie", to be matched against a value stored with the retrieved state information; this protects against delivery of data units to the wrong place.

- The **application information** describes the function of the programs using τ. For existing TCP/IP applications, the value of the identifier is the well-known TCP or UDP port number. Certain bits in the application ID indicate particular attributes of the application, such as *multicast*, whether it is a *well-known* application, whether it is an *encrypted* application, etc. The application ID may be used together with the SRI to speed the process of retrieving state information. For example, in large servers, different methods might be used to organize and retrieve state information for applications with very different packet arrival characteristics.

4.1.2 The Configuration Information

The **Configuration Information** is also made up of two parts: the encryption and predefined configuration information and the data location and header count information. Each one of these provides enough information for τ to determine what protocol headers are present. The predefined configuration information is used when the connection is using a predefined configuration, while the data location and header count information are used when the connection is using an explicit configuration.

- The **encryption and predefined configuration information** allows for determination of the type of encryption used as well as for rapid identification of the set of protocol headers present in the data unit. The encryption field consists of an 8-bit flag that conveys information to τ about the encryption of the message. The configuration information fields consist of two bitmaps, each indicating the presence or absence of up to 24 protocol headers (up to 12 data-touching headers plus up to 12 control headers). The purpose of this field is to support *optimized implementations*, which can use it to quickly select a processing path tailored to the specific information present in this message (e.g., an integrated loop for handing multiple data-touching functions).
- The **data location and header count information** tells the number of control protocol headers and the number of data touching protocol headers that follow the metaheader, and where the user data portion of the data unit begins. This information is only necessary when no predefined configuration is being used.

4.1.3 Header Descriptors

If the predefined configuration field is zero, and the total number of headers is nonzero, one or more **header descriptors** must follow the metaheader. Each such descriptor includes an identifier, which tells τ which protocol function the header should be dispatched to, and a length. The list of header descriptors (or the predefined configuration field, if nonzero) defines a total order on the protocol functions that must process the message. For control functions, this order does not constrain the order they are applied at the receiver. Data-handling functions, however, *must* be applied in the order indicated by the metaheader. For example, it is very important to be consistent about whether a checksum is performed before or after an encryption function.

4.2 Protocol Functions

Protocol functions in τ fall into three classes: those which fit the generic protocol model, those which do not fit the model and have some effect on the "bits on the wire", and those which do not fit the model but are handled via special "intra-τ" interfaces. We next present examples of each type.

4.2.1 The "Generic" Protocol Function

We differentiate between two different types of generic protocol functions: those that touch the data, and those that do not, i.e. perform control functions. The use of a small number of generic interfaces to protocol functions gives τ the ability to quickly and efficiently handle all protocol functions in the same way. There are three different types of events that can cause a protocol function to be called: Incoming Message, Incoming Request and Auxiliary Event (e.g. timer expiry). In each case, a specific entry point is used depending on the state of the connection. The following are some of the possible states that a connection could be in when the event occurs.

- New State: there is a new connection that requires a new state block to be set up and maintained for all protocol functions that are involved in connection set-up.
- Initialize: there is a new connection that requires a new state block to be set up and maintained.
- Existing State: this is an old connection with existing state.
- No State: this a connection that does not require state to be maintained.
- Abort Connection: this is called when the connection is being taken down. The protocol function is responsible for deallocating its own state block.

To support the logical execution of protocols in parallel, a *commit phase* (see figure 3) is used. Since each protocol may begin processing before the outcome of others is known, it is necessary to be able to *abort* a protocol function. The goal of the commit phase is to keep the processing fast, since this processing will be done along the fast path. In order to limit the amount of processing overhead associated with a message that does not commit, we take an optimistic point of view. Each protocol function saves some amount of old state information. If the message is ok, there is no overhead for a commit. If there is a problem, the protocol functions are called through the ABORT_MESSAGE entry point. They can then take the saved information and roll back to their previous state. This removes the responsibility of deciding if and what degree of roll back is needed from τ. The protocol functions decide how much and what state information they need to save in order to be able to rollback. τ provides the mechanism, while the protocols themselves decide the policy.

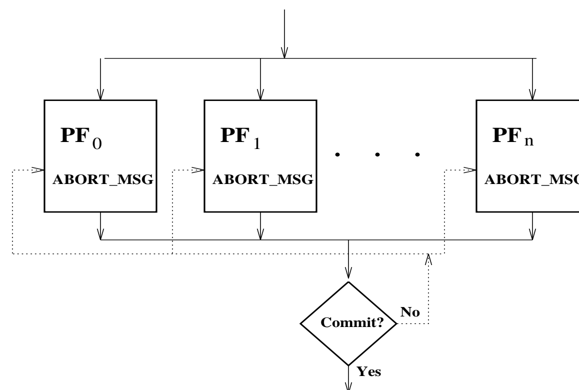

Figure 3: Commit Phase for Protocol Functions

4.2.2 Protocol Functions Affecting the Metaheader

Despite the desire to keep all interactions with the protocol functions as generic as possible, there are some specific protocols that τ needs to know about. Information about these protocols can be determined through a number of different mechanisms, most of which involve the metaheader. The first is information that is directly encoded in the application ID. Since the application ID determines how an incoming data unit is processed, this allows for a very flexible approach. The next way is information that is passed in the configuration section of the header. This is somewhat more restrictive, but still allows a good deal of freedom. The information in the predefined configuration section of the header is broken up into three parts. The first is the encryption field. The rest is the bit maps used to determine which control and data protocols are used in a predefined configuration. τ can also determine information about a protocol from the Header Descriptor for that protocol. The following describes some of the more important protocols that τ needs to interact with and also shows the different types of information exchanged.

Some protocol functions are special in that they effect the determination of what information goes in the metaheader.

Encryption : In order to be able to provide secure communication, τ must provide the ability to use encryption. Unfortunately, the use of encryption slows down the performance of any high speed protocol because the packet must be decrypted before any processing can be done. This results in a tradeoff between security and speed. Since different applications will require different levels of security, τ will provide different possibilities for encrypting messages. τ itself does not specify what type of encryption can be done. That is specified by the application which uses some encryption protocol function. τ simply provides the application the choice of the level of encryption of its messages.

By using τ, the application has a choice of five different levels of encryption:

1. None. No encryption is requested.
2. Data Only. In the simplest case, the application may only want to have the data section secure. This would only require τ to encrypt the data section, leaving the header and the meta-header alone. Although this is the least secure, it is also the quickest. If data only encryption is used, the encryption is handled by a protocol function and is not differentiated in τ from any other protocol function. This level of encryption has the drawback that headers and header information are sent in the clear.
3. Data and Headers. Protocol headers and data are encrypted separately. When the encryption of headers is involved, τ must know how to decrypt before it can do the Demux-and-Dispatch. This information is provided for τ in the encryption bits of the header. We can provide this level of encryption by leaving the meta-header descriptor unencrypted. This leaves τ with the information of where the data section starts as well as any necessary information about predefined configurations. τ can use this information to first decrypt the header descriptors and protocol headers, process the protocol headers and then finally decrypt the data section. Although this will increase security, this level of encryption has the problem that an intruder will be able to determine which part is the header and which part is the data as well as any information about predefined configurations.
4. Data, Headers and Configuration Information. This is similar to the encryption of data and headers only. In this case, however, the predefined configuration information and the data and header location information are encrypted as well, although separately (see figure 4). τ can then decrypt this information first, and be able to start on the headers without having to decrypt the entire message.

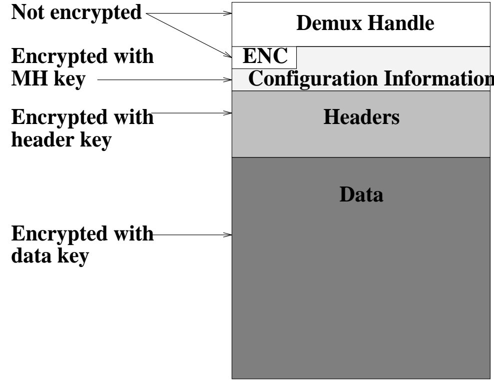

Figure 4: Separate Encryption of Metaheader, Headers and Data

5. The Entire Message. In order to be able to hide the information in the τ header as well, τ needs to provide for an entire τ data unit to be encrypted. This is supported recursively, through the use of well-known application identifiers (see figure 5). For example, if public key encryption is used, the application ID can have a well-known associated key; every data unit arriving at that application ID is first decrypted with the corresponding private key. The result is an entire τ message. There are no restrictions on the make-up of this message. It may contain one of the previously mentioned encryption methods as well. Of course, there is a performance penalty associated with this approach, but it offers increased security to applications that need it.

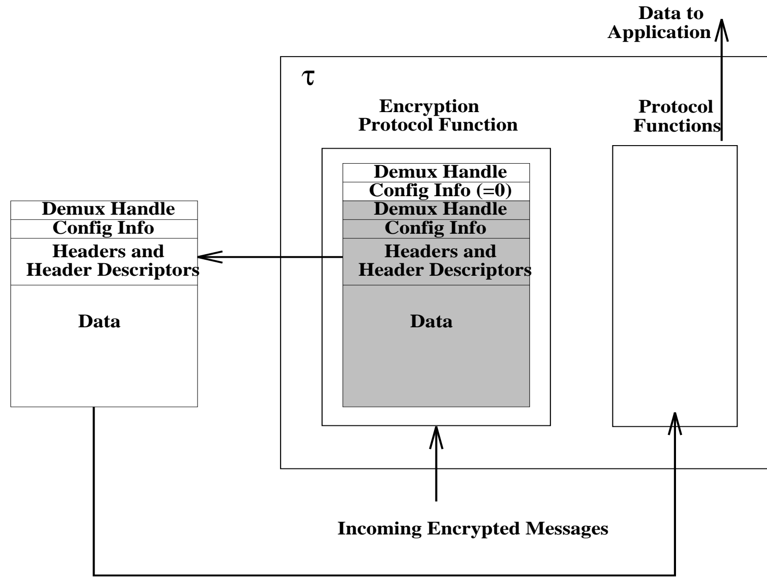

Figure 5: Recursive use of τ for Encryption

Fragmentation and Reassembly : Fragmentation and reassembly are handled similarly to encryption. Any message that is involved in fragmentation or reassembly is sent to a specific application ID (see figure 6). As we know, τ cannot begin processing the message until the entire message has been received. But since each fragment is itself a τ message, certain protocol functions, such as

acknowledgment and retransmission for the fragments, can be performed on the fragments themselves as they are being collected. Once an entire message has been reassembled, the reassembly protocol function passes it on to be handled like a regular message. Note that τ allows a choice of whether other functions required by the application are applied to the original user data unit, or to the *fragments*; that is, whether the fragmentation is done "above" or "below" the other functions. The choice of which is more appropriate will in general depend on the characteristics of the underlying network service.

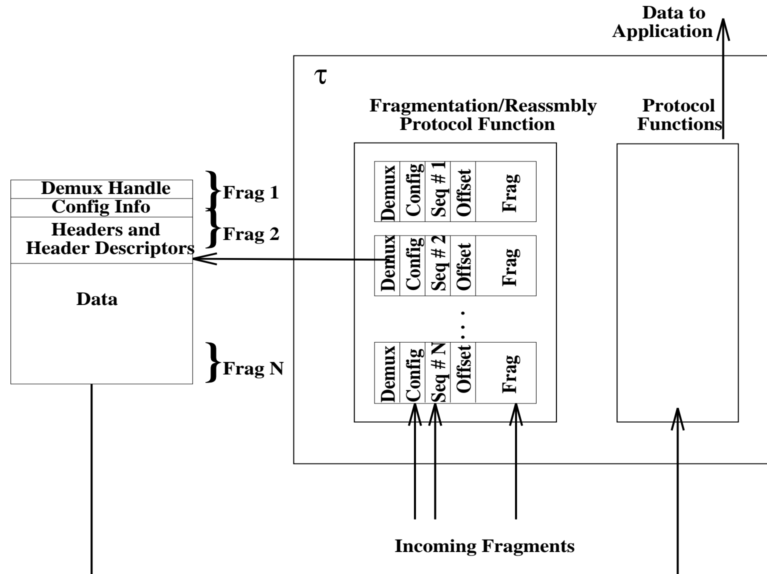

Figure 6: Recursive use of τ for Fragmentation

Reply-To and Data Transfer There are two protocol functions that τ can find out about specifically in the data and control predefined configuration section. One bit in each section is reserved for a specific protocol function (see figure 7). In the control section, this bit is reserved for the "reply-to" protocol function. The "reply-to" information is the Demux handle corresponding to the endpoint to which any reply should be sent. If the bit is set, the "reply-to" protocol function is present. In the data section, the bit is reserved to indicate if there is data present in the message. This is the only way that τ can determine if there is data present in the message when predefined configurations are used. This also allows τ the ability to handle the transfer of data to the application as a data handling protocol function, in particular to integrate it with other data handling functions.

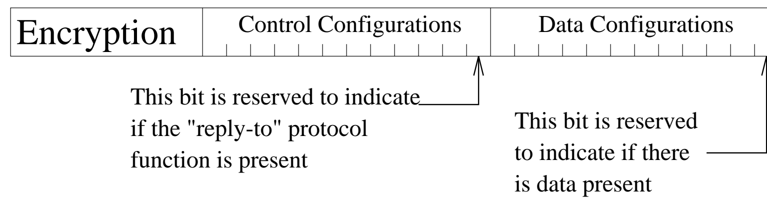

Figure 7: Predefined Configuration Section

4.2.3 Other Non-generic Protocol Functions

Certain other protocol functions affect the internal structure of τ, but don't show up specifically in the tau metaheader. The idea is to allow many different policies for these functions, so τ provides generic mechanisms to interact with them.

Data Flow Control : The model of data flow control is that a request to send out a data message will never violate the constraints of flow control. In other words, τ will never start processing a message unless that message can be sent out immediately. Upon the receipt of a data transfer request, τ will check to make sure that the amount of data is not greater than the amount available to send out. τ gets updates from the flow control protocol function through the use of the global attributes. As the available level for flow control changes, the flow control protocol updates the location. When τ sees that the message will no longer violate the limit, it can start to process the data and send it out.

Message Retransmission : τ provides two possibilities for message retransmission. Due to the overhead of storing processed data, while waiting to determine if a message has been successfully received, τ can keep a pointer to the application's data. This requires that the application keep the data around until signaled by τ that the data has been successfully transmitted. If the message needs to be retransmitted, the message is reprocessed, going through both control and data protocols (see figure 8). If it turns out that the application requires that the data be released immediately or connection is very lossy and the overhead of reprocessing the data is worth the trade off of the amount of space it would take to save the processed data, τ also offers the option of saving the processed data. In this case, the message need only pass through the associated control protocols before it could be sent out.

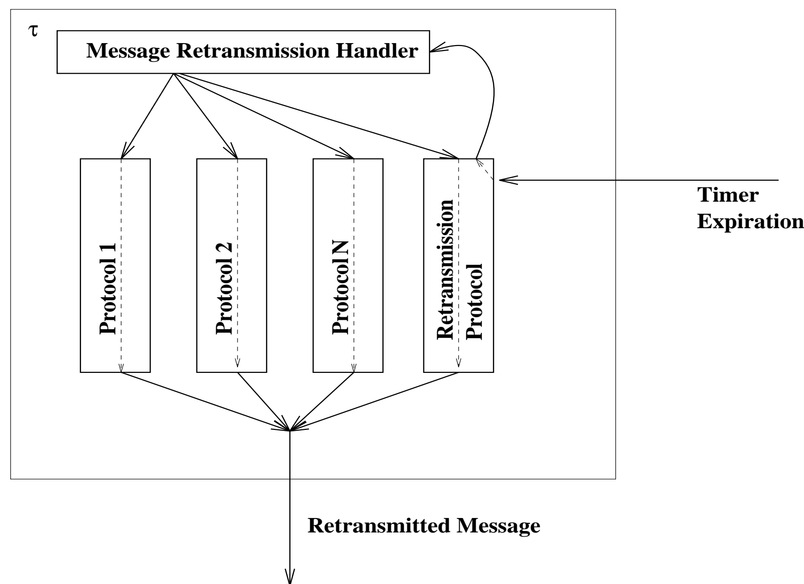

Figure 8: Message Retransmission

5 Concluding Remarks

We have presented a brief overview of the design and some features of τ, a transport protocol framework supporting both high performance and extensibility. The protocol provides a mechanism for flexible composition of functionality without layering, and supports efficient, nonlayered multiplexing. It can provide a range of services, from simple best-effort datagram to reliable byte stream, including a variety of options for encryption.

We are currently implementing τ. Our initial goal is a datagram service capable of throughput rates comparable to those of decent UDP [Pos80] implementations. We plan to add functions to τ to support a variety of applications, including multimedia and distributed simulation.

References

[AP93] M. B. Abbott and L. L. Peterson. Increasing network throughput by integrating protocol layers. *IEEE/ACM Transactions on Networking*, 1(5):600–610, October 1993.

[Bjö94] Mats Björkman. The xx-kernel parallel protocol execution environment. Technical Report Draft, Department of Computer Science, Uppsala University, Sweeden, July 1994.

[Cal93] Kenneth L. Calvert. Beyond layering: Modularity considerations for protocol architectures. In *Proceedings International Conference on Network Protocols, San Francisco*, October 1993.

[CT90] David D. Clark and David L. Tennenhouse. Architectural considerations for a new generation of protocols. In *ACM SIGCOMM-1990 Symposium*, pages 200–208, September 1990.

[Fel90] David C. Feldmeier. Multiplexing issues in communication system design. In *Proceedings ACM SIGCOMM '90 Symposium, Philadelphia*, pages 209–219, September 1990.

[Fel93] David C. Feldmeier. A framework of architectural concepts for high-speed communication systems. *IEEE Journal on Selected Areas in Communications*, 11(4):480–488, May 1993.

[FRS93] M. Fry, A. Richards, and A. Seneviratne. Framework for implementing the next generation of transport protocols. In *Proceedings Fourth International Workshop on Network and Operating System Support for Digital Audio and Video, Lancaster, England*, 1993.

[fS86] International Organization for Standardization. Connection-oriented transport protocol specification, 1986.

[Haa91] Zygmunt Haas. A protocol structure for high-speed communication over broadband ISDN. *IEEE Network Magazine*, pages 64–70, January 1991.

[LKAS93a] Bert Lindgren, Bobby Krupczak, Mostafa Ammar, and Karsten Schwan. An architecture and toolkit for parallel and configurable protocols. In *Proceedings of the International Conference on Network Protocols (ICNP-93)*, September 1993.

[LKAS93b] Bert Lindgren, Bobby Krupczak, Mostafa Ammar, and Karsten Schwan. Parallelism and configurability in high performance protocol architectures. In *Proceedings of the Second Workshop on High-Performance Communications*, September 1993.

[NYKT94] Erich M. Nahum, David J. Yates, James F. Kurose, and Don Towsley. Performance issues in parallelized network protocols. In *Proceedings of the First Symposium on Operating System Design and Implementation*. Department of Computer Science, University of Massachusetts, 1994.

[OP92] S. W. O'Malley and L. L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10:110–143, May 1992.

[Pos80] J. Postel. *User Datagram Protocol*. Network Information Center, Internet Request for Comments No. 768, August 1980.

[PPVW93] Thomas Plagemann, Bernhard Plattner, Martin Vogt, and Thomas Walter. Modules as building blocks for protocol configuration. In *Proceedings of the International Conference on Network Protocols (ICNP-93)*. Swiss Federal Institute of Technology Zurich, September 1993.

[PS91] Thomas F. La Porta and Mischa Schwartz. Architectures, features, and implementation of high-speed transport protocols. *IEEE Network Magazine*, 4(2):14–22, May 1991.

[PS93] Thomas F. La Porta and Mischa Schwartz. The multistream protocol: A highly flexible high-speed transport protocol. *IEEE Journal on Selected Areas of Communications*, 11(4):519–530, May 1993.

[SBS93] Douglas C. Schmidt, Donald F. Box, and Tatsuya Suda. Adaptive: A dynamically assembled protocol transformation, integration, and evaluation environment. *Journal of concurrency: Practice and Experience*, June 1993.

[Ten89] David L. Tennenhouse. Layered multiplexing considered harmful. In *Proceedings of IFIP Workshop on Protocols for High-Speed Networks*, pages 143–148, May 1989.

[Zit91] Martina Zitterbart. High-speed transport components. *IEEE Network*, January 1991.

[ZT93] Martina Zitterbart and Ahmed N. Tantawy. A model for flexible high-performance communication subsystems. *IEEE Journal on Selected Areas of Communications*, 11(4):507–518, May 1993.