Chapter 3

THE UNIVERSAL TRANSPORT LIBRARY (UTL)

3.1 Introduction

This chapter describes the Universal Transport Library (UTL) developed by the author as a tool for investigating flexible Transport QoS in general, and PO/PR transport protocols in particular. The author designed the core architecture of this transport layer software, and completed a substantial portion of the implementation. Two MS students, Ed Golden and Mason Taube, provided design consultation, programming and debugging support under the author's supervision.

UTL is a library that can be linked in with an application, to provide a range of transport services through a single API. The transport services provided in UTL include simple wrappers for TCP and UDP, as well as a range of PO/PR transport services. For developers of transport layer services and protocols, UTL provides a framework for rapid prototyping of transport layer implementations. For application writers, UTL provides the ability to easily compare an application's performance over a wide range of transport protocols. The implementation of UTL used in this dissertation is for Solaris 2.6; an implementation for Linux has also been completed.

3.1.1 Organization of this chapter

This chapter is organized as follows. Section 3.2 first describes the motivation for UTL, namely certain problems that we encountered with comparing the performance of ReMDoR over various transport protocols. Section 3.2 goes on to describe how UTL addresses these problems. Section 3.3 provides an executive summary of UTL; this section is an overview of the most important aspects of UTL that should be understood before reading the performance experiment results of Chapters 5 through 7.

Sections 3.4 through 3.7 describe UTL in detail; readers primarily interested in performance results may wish to skip these sections on first reading:

- Section 3.4 provides a formal specification of the rules for composing UTL mechanism from layers, and determining the resulting QoS.
- Section 3.5 describes a few of the key design decisions that faced the developers of UTL, and provides the rationale for the design choices made.
- Section 3.6 highlights a few protocol details for KXP, KX2 and KX3; these protocols are basis of the key transport services provided by UTL.
- Section 3.7 describes the means by which UTL was tested and debugged.
- Section 3.8 describes related work, including work on implementing protocols at user level, and an overview of the *x-Kernel* (Hutchinson and Peterson, 1988) which provides a similar protocol framework.

Finally, Section 3.9 summarizes the material presented in this chapter.

3.2 Motivation

Suppose we want to compare a PO/PR transport service such as POCv2 to traditional transport services: e.g., ordered/reliable service, and unordered/unreliable service. We might imagine that we could compare POCv2 to TCP and UDP. However, it turns out that TCP and UDP differ in many ways other than order and reliability. Here are just three examples:

- TCP is connection oriented, while UDP is connectionless. Connection-oriented communication requires a different set of system calls to set up communication, and clean up afterwards.
- TCP is byte-stream oriented, while UDP is message-oriented. To achieve message-oriented communication over TCP, a considerable amount of extra code must be added to the application.
- TCP provides flow control and congestion control; UDP does not.

If we want to write an application that can operate over both TCP and UDP, as well as experimental protocols such as POCv2, we will likely have to write a great deal of special-case code. Special-case code that depends on specific transport layers is unappealing for several reasons. First it is time-consuming and error prone. Second, writing special-case code makes experimentation with additional transport services difficult, since with this approach, each time we want to add a new transport service to the experiment, the application must be modified. Finally, it opens the experiments up to criticism that the comparison is unfair, since the application code being executed depends significantly on the transport protocol.

Therefore, our first goal in developing UTL is to eliminate (or at least, greatly reduce) the need for special-case code in the application. An application developer using UTL should only have to write the communications part of an application one time, and the application should then work in a sensible fashion

regardless of the transport protocol being used. The vision is that an application specifies the transport protocol only *once*. The protocol is specified as a parameter to the function that listens for connections (in the case of a server) or the function that requests a connection (in the case of a client.)

Now clearly, there are times when some services are inappropriate for some applications. An application that inherently requires a reliable service—for example, a banking application—cannot be expected to perform correctly over an unreliable service. UTL does not promise that all applications will run correctly over all protocols. The application designer is responsible for restricting applications to a subset of UTL services that are appropriate for the needs of that application.

What UTL does promise is that (1) the basic transport layer functions of read, write, connect, listen, accept, and so forth, will take the same parameters regardless of the transport service selected, (2) that each of these functions will behave in a manner consistent with the service selected. For example, if an application does a `write()` operation over a UTL service that is reliable, UTL guarantees that either the message is delivered, or else the application will receive a notification that the message may not have been delivered. If the application does a `write()` with a UTL service that is unreliable, the message may or may not be delivered.

In summary, the first principle of UTL is that except when selecting the transport service on the initial listen or connect operation, the application need never be concerned with what transport service is being used at any given time.

3.3 Overview of UTL

This section provides the reader with a quick introduction to the most important aspects of UTL. It is especially appropriate for the reader who wants just

enough understanding of UTL to be able to interpret the performance results in their proper context. Nearly all the issues discussed in this section are covered in more detail later in this chapter, or elsewhere in the dissertation.

3.3.1 Central principles of UTL

Several *central principles* guided the design and implementation of UTL. Here, we merely list them with a brief explanation; each is covered in more detail later in the chapter.

(1) Avoidance of protocol[20] specific, special-case code in the application. (Explained in Section 3.2 above).

(2) Application level framing. Following the example of (Clark and Tennenhouse, 1990), UTL follows the philosophy that transport and application layers should cooperate to preserve ADUs as atomic entities, and provides appropriate reliability and order for individual ADUs where possible.

(3) Reasonable fallbacks. This principle is a corollary of the "no-special-case-code" principle. If an application requests an operation that cannot be performed by the specified mechanism, UTL will do the best it can. For example, if a partial order is requested when the mechanism only supports unordered service, the protocol will still allow data transfer, albeit providing unordered service.

---

[20] After we define the term "mechanism" in Section 3.3.2, we will cite this principle as "avoidance of mechanism-specific code" rather than "avoidance of protocol-specific code."

(4) Minimizing Data Copies. Minimizing data copies is crucial, since performance is of the essence for any application using UTL.

3.3.2 Common service model: connection oriented, PO/PR message service

UTL provides an API similar, though not identical, to the Berkeley Sockets API. While the purpose of UTL is to provide a diversity of transport services, to meet the goal of providing a common API, UTL is based on a common service model of *connection-oriented, message-oriented* service. Section 3.5.2 explains the rationale for this choice in more detail.

In addition, since PO/PR is the most general service in terms of order and reliability, the common service model assumes PO/PR service. Therefore, operations to modify the service profile[21] are included in the basic API of UTL. However, based on the reasonable-fallbacks principle, since the default service profile is for a single reliable object per period, UTL applications are free to ignore the partial order and partially reliability features of UTL. If the only services required for a given application or experiment are combinations of (ordered vs. unordered) and (reliable vs. unreliable) service, the API functions specific to partial order and partial reliability never need to be called.

3.3.3 Connection-oriented implies three phase operation—nothing more

While the term *connection-oriented (CO)* is often associated with ordered/reliable service, it should be emphasized that CO does not imply either order

[21] See Section 2.4.3 for a discussion of the term “service profile” as it pertains to a PO/PR service.

or reliability. CO refers only to the fact that there are three phases of operation: (1) connection establishment, (2) data transfer, (3) connection teardown. (A more detailed discussion of this point can be found in (Iren et al., 1999a)). Though all the transport services provided by UTL are CO, the levels of reliability and order range from unordered/unreliable to ordered/reliable, with many gradations in between.

For CO transport protocols, a consequence of three-phase operation is that state information is always maintained at both sides of the connection—at least, to indicate the current phase of operation. More typically, state variables are kept to manage the provision of features such as reliability, order, flow control, and so forth. Within UTL, this state is maintained in an abstraction called a *session.* Just as in the Berkeley Sockets API, an application references a session using a *file descriptor.* A file descriptor is a positive integer that corresponds exactly to the file descriptor used by Unix for the underlying UDP or TCP socket that provides service to the UTL session.

3.3.2 Selecting transport QoS via UTL mechanisms

As of UTL version 0.90, there were 34 different transport services provided, as shown in Table 3.1. For reasons explained in Section 3.5.8, we refer to each of these services as a *mechanism*. Each mechanism has a 2 or 3 character name consisting of letters and digits name; the names should be regarded as mnemonic codes rather than abbreviations or acronyms. To use a transport service provided by UTL, an application must either passively listen for a connection (the usual case for servers), or actively connect to a listening application (the usual case for clients). The `utl_Listen()` and `utl_Connect()` functions used for this purpose require that one of the mechanisms in Table 3.1 be specified. The mechanism specified in these

function calls is the primary means by which the application specifies the desired transport service, protocol, and QoS for a connection.

Typically, an application based on UTL provides either a command line option or a menu to the application user by which a UTL mechanism can be selected, as shown in Figure 3.1. Because of the no-special-case-code design principle, an application using UTL can simply

(1) take a character string representing a UTL mechanism name,

(2) convert it to a UTL mechanism number (via the `utl_StringToMech()` function)

(3) pass the UTL mechanism number to the `utl_Connect()` or `utl_Listen()` function.

After the `utl_Connect()` or `utl_Listen()` returns, the application uses the connection in the same manner regardless of the mechanism being used.

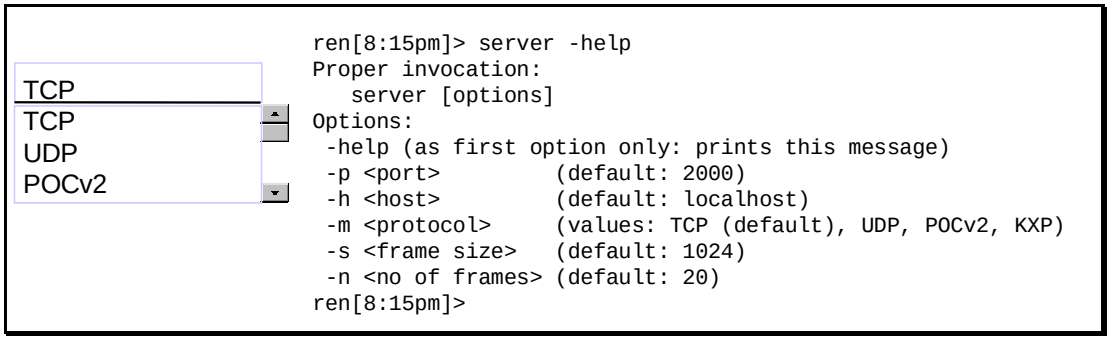

Figure 3.1 Selecting from among different transport protocols by a menu (left) or command line option ("-m" for "mechanism", right).

3.3.3 Modifying transport QoS via UTL protocol parameters

In addition to selecting a mechanism, the UTL service user can modify any of several optional protocol parameters to further tailor the transport service and/or protocol. For example, some UTL mechanisms allow the application to:

- modify the sending and receiving window sizes,
- modify the initial values used for RTO estimation,
- turn on and off various aspects of congestion avoidance features such as slow start, and fast-retransmit.

Table 3.3 shows the protocol parameter values that are supported in UTL version 0.90. In keeping with the no-special-case-code principle, any of these protocol parameters can be set for any mechanism. Based on the reasonable-fallbacks principle, if the protocol parameter is not applicable to or incompatible with the mechanism selected, a reasonable alternative action will be taken—usually, doing nothing.

Table 3.1 UTL mechanisms

Mech	UTL QoS Parameters (see Table 3.2 for explanation of values)						Layers (bottom to top)
	Order	Rel.	Dupl.	Expl.Rel Sync.	Cong. Avoid	App/Tr Flow Ctrl	
	U	PRk	Y	N	N	S	KXP
TX	O	R	N	N	T	S	TXL
UC	U	U	Y	N	N	S	KXP
SP	O	R	N	N	N	S	KXP,TOL
POC	PO	R	N	Y	N	S	KXP,POL
PT	O	R	N	Y	T	S	TXL,POL
NX	U	PRk	Y	N	N	S	KXP,NUL
NT	O	R	N	N	T	S	TXL,NUL
X2	U	PRk	Y	N	2	S	KX2
PTX	O	R	N	Y	N	S	KXP,TOL,POL
NTX	O	PRk	N	N	N	S	TX ,NUL
POS	PO	R	N	Y	N	S	KXP,POL
PO2	PO	R	N	Y	2	S	KX2,POL
PS2	PO	R	N	Y	2	S	KX2,POL
T2	O	R	N	Y	2	S	KX2,TOL,POL
T3	O	R	N	Y	3	SR	KX3,TOL,POL
R2	PO	R	N	Y	2	S	KX2,NUL,POL
R3	PO	R	N	Y	3	SR	KX3,NUL,POL
SP2	O	R	N	N	2	S	KX2,TOL
X3	U	PRk	N	N	3	SR	KX3
PO3	PO	R	N	Y	3	SR	KX3,POL
T3	O	R	N	Y	3	SR	KX3,TOL,POL
SP3	O	R	N	N	3	SR	KX3,TOL
N2	U	PRk	Y	N	2	S	KX2,NUL
N3	U	PRk	N	N	3	SR	KX3,NUL
X2E	U	PRk	Y	N	N	S	KX2
X3E	U	PRk	N	N	N	SR	KX3
N2E	U	PRk	Y	N	N	S	KX2,NUL
N3E	U	PRk	N	N	N	SR	KX3,NUL
P2E	PO	R	N	Y	N	S	KX2,POL
P3E	PO	R	N	Y	N	SR	KX3,POL
S2E	O	R	N	N	N	S	KX2,TOL
S3E	O	R	N	N	N	SR	KX3,TOL
T2E	O	R	N	Y	N	S	KX2,TOL,POL
T3E	O	R	N	Y	N	SR	KX3,TOL,POL
R2E	PO	R	N	Y	N	S	KX2,NUL,POL
R3E	PO	R	N	Y	N	SR	KX3,NUL,POL
SR	U	PR	Y	N	N	S	KXP,SRL

*SRL is the segmentation/reassembly layer, not yet implemented as of version 0.90.

Table 3.2 UTL QoS Parameters (legend for Table 3.1)

To help the reader understand Tables 3.1, the codes for each QoS parameter are positioned differently in the column, corresponding roughly to the level of service provided:

- entries positioned toward the left of the value column do *more* work to enhance of the underlying network QoS
- entries positioned towards the right do *less* work
- entries lying between the two extremes are positioned accordingly.

UTL QoS Parameter	Values	Explanation	see Section
Order **O** ={O,PO,U}	O	**ordered**: messages delivered in exact sequence submitted by sender	2.1
	PO	**partially ordered**: partial order governs message delivery	
	U	**unordered:** no resequencing of out-of-order messages is done	
Reliability **R** ={R,PRk, PR2,K,U}	R	**reliable:** all messages delivered, or connection is aborted	
	PRk PR2	**partially reliable:** varying reliability guarantees for each message. PRk: partial reliability as in KXP: individual messages may be given *k* values, where *k* is the number of transmissions (0 indicates infinity, fully reliable) PR2: partial reliability as in POCv2 (See Section 2.8)	2.1
	K	***k*-xmit reliable**: no delivery guarantees, msgs retransmitted *k*-times, where *k* is fixed for the lifetime of the connection.	
	U	**unreliable:** no delivery guarantees	
Duplicates **D**={N,Y}	N	**no-duplicates**; each message delivered at most once	2.1
	Y	**maybe-duplicates**: duplicate messages may be delivered	
Congestion Avoidance **C**={T,3,2,N}	T	**TCP-congestion-avoidance**: TCP actually used (not an emulation)	3.3.7
	3	**KX3**: emulation of TCP congestion avoidance, including slow start, cwnd, and fast retransmit (work in progress.)	
	2	**KX2:** emulates only TCP slow start and cwnd; no fast retransmit	
	N	**none**: traditional sliding window flow control only.	
Explicit Release **E**={Y,N}	Y	**yes**: explicit release synchronization is provided	2.6
	N	**no**: explicit release synchronization is not provided	
***streamEnd* Default** **S**={1,0,n/a}	1	**1***: streamEnd* is on by default: each write produces a separate object	2.3
	0	**0**: *streamEnd* is off by default: each write appends to the current object	
	n/a	**n/a**: not applicable; stream abstraction not supported by mechanism	
Application/ Transport Flow Control **A**={SR,S,N}	SR	**sender and receiver:** application sender can be throttled to enforce a finite buffer size for both (a) queued outbound data at transport sender, and (b) queued inbound data at transport receiver.	3.5.9
	S	**sender only:** application sender can be throttled to enforce a finite buffer size for queued outbound data at transport sender, but the amount of queued inbound data at transport receiver can arbitrarily large, limited only by available heap space.	
	N	**none**: app-tr transport flow control is not done: inbound and outbound queues at the TSAP may grow without bound as heap space permits.	
Layers	see Table 3.4 and Table 3.5		3.3.5, 3.3.6

Table 3.3 UTL protocol parameters

The index column one is used in the formal definition of a UTL protocol parameters vector (Section 3.4.1).

index	Protocol Parameter	Data Type (units)	Explanation
1	maxXmits	unsigned integer	Maximum number of times a packet will be transmitted; 0 indicates infinity (reliable service).This is the *k* value as in the *k*-xmit reliability defined in (Marasli, 1997b). Used in KXP, KX2, KX3.
2	enableCongestionAvoidance	Boolean	Enables or disables the congestion avoidance features of the KX2 and KX3 layers)
3	serviceProfile	array of unsigned integers	Partial order and reliability vector for PO/PR service. Used only by POL. (See Section 2.4.3)
4	explicitRelease	Boolean	For PO service, indicates whether successors of delivered objects should be released immediately, or only when the application specifically indicates to do so. (used for coarse-grained synchronization) Used only by POL.
5	localReceiveWindow	unsigned integer (bytes)	When app-transport flow control is used at the receiver (as in KX3), indicates the number of bytes that may be buffered for delivery at the receiver before the flow is halted
6	localSndWindow	unsigned integer (messages)	Size of sending window for ordinary window-based flow control between sending and receiving transport entity; upper bound on number of outstanding unacknowledged messages. Used only in KX2, KX3.
7	RTOInitialAverage	unsigned integer (ms)	Initial value for estimate of mean RTT in Van Jacobson's formula for computing round-trip times[22] Used in KXP,KX2,KX3
8	RTOInitialDeviation	unsigned integer (ms)	Initial value for estimate of mean deviation of RTT in Van Jacobson's formula for computing round-trip times[3]. Used in KXP,KX2,KX3
9	streamEnd	Boolean	should next object written finish a stream object, or continue the current stream object? use only by POL.
10	objNum	unsigned integer	object number used for next `Write()` operation

[22] (Jacobson 1988). For a tutorial presentation of the formula, see (Stevens, 1994).

3.3.4 UTL is a library providing flexible transport QoS, not a QoS Architecture

QoS architectures typically include provisions for an application to request a specific guaranteed QoS from the service provider. When presented with a request, a service provider assesses its own ability to provide the requested QoS, given its current resources and current load. The service provider then either accepts the request, or denies the request (in some cases, this constitutes *admission control*), perhaps making a counter-proposal as in a human negotiation. A connection is only established when the negotiating entities can agree to a QoS contract. (Aurrecoechea et al., 1998) provides a survey of QoS architectures that fit this general model.

UTL, by contrast, makes no attempt to negotiate, or guarantee minimum service levels if the application chooses a mechanism that is inappropriate for a given application. The logical consequence of the no-mechanism-specific-code and reasonable-fallbacks principles is that with UTL, the application always gets exactly the protocol that the application requests, even if that protocol does not meet the application's qualitative needs. Quantitative QoS guarantees are, of course, impossible since UTL assumes an underlying unreliable network that can make no guarantees. Thus, the QoS provided by UTL is based only on the mechanism selection, the protocol parameter values chosen, and the QoS of the underlying network.

While not all protocol parameter settings are appropriate to all applications, an application would typically *not* restrict the user from setting whatever protocol parameters she or he chooses, even if such settings would have no effect. In some cases, UTL will return error codes to indicate that a certain protocol parameter setting is not supported for a particular mechanism, for example, if a service profile is offered to a mechanism that does not support partial order. However, as we explain

further in Section 3.5.6, UTL does not close a connection if inappropriate protocol parameter choices are made.

3.3.5 Mechanisms are composed of layers

Table 3.1 also shows that (with the exception of the RAW mechanism, explained in Section 3.5.8) UTL mechanisms are composed from UTL *layers*. Tables 3.4 and 3.5 summarize the layers in UTL, while Figure 3.2 shows a subset of the UTL mechanisms, illustrating the configuration of the various UTL layers of which each is comprised. Each UTL layer is essentially a sublayer of the transport layer, and operates according to the usual layered architecture principles. That is, each UTL layer:

- provides a service to the layer above,
- by utilizing the services of the layer below,
- to exchange PDUs with its peer layer.

Each layer operates according to its own protocol, and has its own protocol header. As per the usual practice, upper layers encapsulate their PDUs in lower layers. For example, we note that in Table 3.1, the T3 mechanism is composed of the KX3, TOL and POL layers. Figure 3.2 illustrates the positions of the various headers on a message sent via the T3 mechanism. Thus a clear box view of UTL is that the T3 mechanism is a transport service composed of three layers, each with its own protocol. By contrast, a black box view of UTL would see T3 as a single protocol, where the T3 header consists of the concatenation of the headers of its constituent layers, from bottom to top (KX3, TOL, POL).

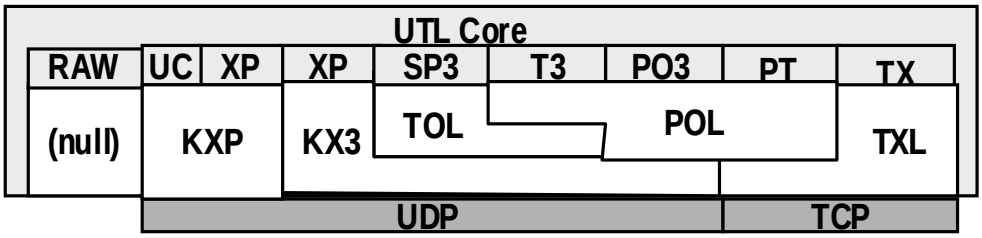

Figure 3.2 UTL mechanisms composed of layers

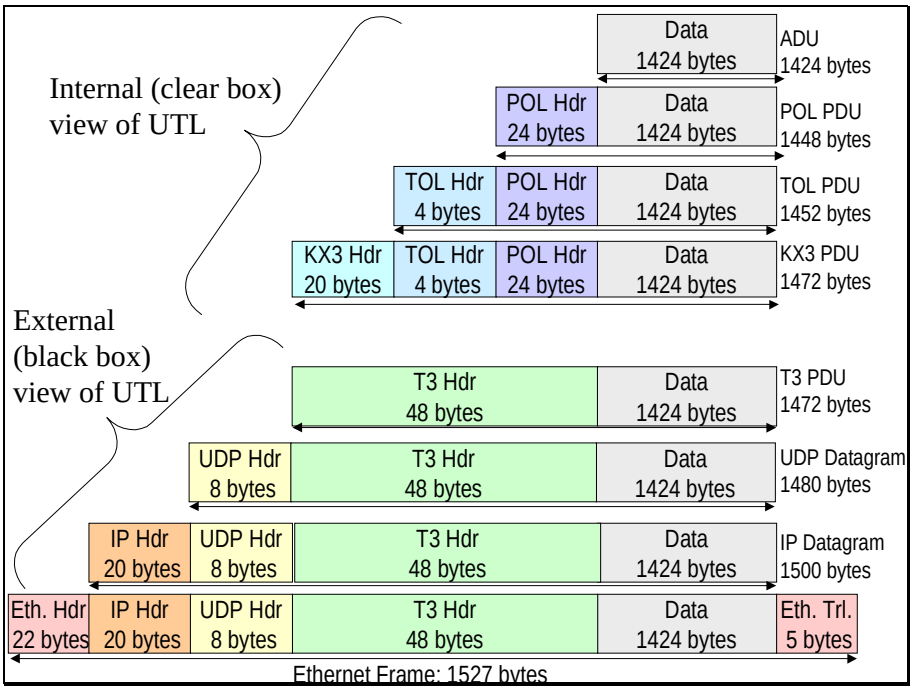

Figure 3.3 UTL layer encapsulation example: T3 over Ethernet

As explained previously in Section 3.3.1, the first principle of UTL is no mechanism-specific code. It turns out, that just as there is no mechanism-specific code in the applications that run over UTL, in fact there is also no mechanism-specific code

within the implementation of UTL. A UTL mechanism is specified only by a set of protocol parameter values, and a stack of layers. While each *layer* is implemented with specific code, the only code in UTL that refers to specific mechanisms is the initialization of a data structure[23] that encodes Table 3.1. All references to mechanisms in UTL then use this data structure to execute the appropriate functions implemented by each layer making up that mechanism. Adding a new mechanism to UTL is therefore a simple fifteen-minute process of adding a few table entries in the routines that initialize this data structure headers files, and recompiling. (Adding a new layer, by contrast, can be on the order of days, weeks or months.)

3.3.6 Rules for composing mechanisms from layers

Section 3.4 provides a detailed formal description of the rules for composing mechanisms from UTL layers; in this section, we just sketch the main ideas of this framework.

The QoS provided by a UTL mechanism is determined by the layers and protocol parameter values of which it is composed. Each UTL mechanism has exactly one *bottom layer* to interface with the standard transport services TCP or UDP. The bottom layer must one of the layers listed in Table 3.4. In addition, the mechanism may have zero or more additional *upper layers* on top of the bottom layer. The layers that may be used as upper layers are listed in Table 3.4. All UTL layers are required to provide a standard set of services to the layer above and the layer below. This standard layer-to-layer interface allows the layers to be placed in any configuration, as

[23] (programmer's note) specifically, see `the utlMechInfo` array in `utlDefs.h`.

long as the layer below meets the minimum service requirements of the layer above.[24] The service requirements of upper layers are listed in the second column in Table 3.5. Note that since upper layers are not required for a mechanism, any of the so-called bottom layers may also end up being the top layer of a mechanism. For example, the UTL mechanisms TX, XP, UC, X2 and X3 are all single layer mechanisms, where the bottom layer is also the top layer. The fact that the KXP, KX2 and KX3 layers may serve as both bottom layer and top layer for a given mechanism is significant because the top layer of a mechanism has a special responsibility: specifically, the top layer is responsible for communication with the application.

3.3.7 Bottom layers: TXL, KXP, KX2, KX3

Certain layers are specifically designed to interface with either UDP or TCP directly; these layers are referred to as bottom layers. Each mechanism must include exactly one bottom layer. UTL mechanisms are named in the same manner as UTL layers, with three-letter codes that should be regarded as mnemonics rather than acronyms or abbreviations. Table 3.5 provides a list of the bottom layers in UTL v0.90 in the order in which they were first written, and not surprisingly, in ascending order of complexity. Note that TXL is written on top of TCP, while KXP, KX2 and KX3 are all written on top of UDP. The KXx family of layers illustrates one of the benefits of the UTL architecture: KXP, KX2 and KX3 are all iterations of the same basic service, but with major changes to the protocol. Because the UTL API hides the differences among underlying services, it was easy to introduce new versions of KXP

[24] From a design pattern perspective (Gamma et al., 1995), the bottom layers can be seen as examples of the *adapter* design pattern since they convert the underlying UDP and TCP transport services to the layer-to-layer interface required by UTL.

to experiment with innovations: the applications were unaffected. They required only to be recompiled[25] and relinked with the new version of UTL. Furthermore, the upper layer providing partial order and total order (POL and TOL) were entirely unaffected by the changes to KXP as it evolved into KX2 and KX3.

Section 3.6 summarizes some of the details of the KXP, KX2 and KX3 protocols.

3.3.8 Upper layers: TOL, POL, NUL, SRL, and layer stacking rules

The upper layers of UTL are shown in Table 3.5. The role of upper layers in UTL is to provide services such as resequencing out-of-order packets, and segmentation/reassembly. Upper layers can function at the top or in the middle of a mechanism's stack, but may never appear at the bottom. A bottom layer is always required as an adapter between the UDP or TCP interface provided by the operating system, and the UTL layer-to-layer interface. Similarly, since a bottom layer implements UTL's layer-to-layer interface only at its upper Service Access Point (SAP), a bottom layer can never appear anywhere except the bottom of a mechanism.

[25] Recompilation picks up the header files containing the new protocol definitions; no change to the application source code is required.

Total Ordering Layer (TOL)

The TOL layer (total ordering layer) assumes that the underlying layer is reliable. Typically, the layer below TOL is KXP, KX2 or KX3, with an immutable *maxXmits* value of 0 (representing reliable service.) The TOL layer simply adds a 40-byte sequence number and uses it to resequence out-of-order packets.

Partial Ordering Layer (POL)

The POL layer (partial ordering layer) provides partial order service with explicit release synchronization. Future versions will also implement the PR reliability class as defined in Chapter 2. A full implementation of POCv2 will consist of the combination of POL (with the addition of support for PR reliability class) plus KX3.

Table 3.4 Bottom layers in UTL (may also serve as top layers)

The functions referred to in the table and defined beneath it are used in the formal specifications of Section 3.4.

				QoS provided (see Table 3.2 for explanation of values)					
Layer	**Explanation of Mnemonic**	**Built over**	**Enhances underlying service by adding:**	**Order**	**Reliability**	**Duplicates**	**Explicit Release**	**Cong Avoid**	**App/Tr Flow Ctrl**
TXL	TCP-xmission layer[26],	TCP	Message orientation	O	R	N		T	
KXP	k-xmit protocol[27]		Connection-orientation, partial reliability					N	S
KX2	k-xmit protocol version two		Connection-orientation, partial-reliability, and optional slow start/cwnd congestion avoidance			Y		f_2	
KX3	k-xmit protocol version three	UDP	Connection-orientation, partial-reliability, and optional slow start/cwnd congestion avoidance with fast retransmit, application-transport flow control at the receiver.	U	f_1	N	N	f_3	SR

Definitions for QoS functions referenced in Table 3.4:

f_1: if mutable(pv.maxXmits) use PRk
else
{
    if maxXmits is 0, use R
    if maxXmits is 1, use U
    otherwise use K
}

f_2: if pv.enableCongestionAvoidance is true, use 2
        otherwise use N

f_3: if pv.enableCongestionAvoidance is true, use 3
        otherwise use N

[26] "xmission" is pronounced "transmission".

[27] "*k*-xmit" is pronounced "*k*-transmit", and is defined in (Marasli et al., 1996) as partial reliability where a packet is transmitted at most *k* times, and then dropped.

Table 3.5 Upper layers in UTL (may be middle or top layers)

A blank entry indicates a QoS parameter not affected by the layer.

Layer	Explanation of Mnemonic	Built over (minimum service requirements for underlying stack)	Enhances underlying service by adding:	QoS provided (see Table 3.2 for explanation of values)			
				Order	Reliability	Duplicates	Explicit Release
NUL	null layer	any well-formed stack	nothing; a null layer for testing purposes only				N
TOL	total order layer	any reliable[28] stack	totally ordered service	O	R	N	N
POL	partial order layer	any partially-reliable[29] stack	partial order, and partial reliability as in POCv2	PO	PR[30]	N	Y
SRL	segmentation/ reassembly layer	any partially-reliable stack	segmentation/reassembly (not yet implemented)	U		N	N

[28] The KXP, KX2 or KX3 layers can be made reliable, for purposes of this requirement, by setting an immutable *k* value of zero at the mechanism level.

[29] The current implementation of the POL layer requires the underlying stack to be reliable; future work includes a new version of the POL layer that can be placed over a partially-reliable stack so that the full specification of POCv2 can be realized in the UTL framework.

[30] The POL layer does not *add* reliability in the sense of providing for retransmissions or forward error correction. However, if the underlying layer supports cancellation of reliability for individual messages, POL adds a PR reliability class that integrates reliability with partial order and explicit release synchronization.

Null Layer (NUL)

The NUL (null) layer adds no functionality but does serve three purposes:

(1) It provides a means to add four bytes of dummy header so that a comparison of partial order to total order is more fair (we explain this point further in Section 5.1)

(2) It provides a means of evaluating the overhead of having multiple layers in a UTL mechanism e.g., by comparing the performance of

- P2=KX2,POL vs. R2=KX2,NUL,POL, or
- X3=KX3 vs. N3=KX3,NUL.

(3) It provides a tool for helping to isolate bugs in UTL. For example, suppose a certain bug occurs in SP2. Checking whether the same bug occurs in X2 and/or N2 can help the investigator better determine the nature of the bug:

- A bug occurring only with SP2 indicates a TOL problem.
- A bug occurring with SP2, N2 and X2 indicates a KX2 problem.
- A bug occurring with N2 and SP2, but not X2, indicates a problem related to KX2's interface to a layer above.

Segmentation Reassembly Layer (SRL)

The SRL layer is a segmentation/reassembly layer. The original design of POCv2 and UTL called for such a layer to implemented, and the internal data structures of UTL were designed to support segmentation/reassembly. As it turned out, by the time we were ready to begin implementing, the increasing emphasis on Application Level Framing in our project made the notion of segmentation/reassembly somewhat of an anathema. ALF design suggests that all data units should be divided into ADUs that are less than or equal to the Path MTU size, thus avoiding the need to segment/reassemble at the transport layer or below. Thus for the performance investigations in this dissertation and in (Iren, 1999c), the need never arose for a segmentation/reassembly layer in practice. Finishing an SRL layer is part of the future work for UTL.

3.3.9 User level implementation with cooperative multitasking

The transport layer functionality in UTL is implemented at *user level*[31] rather than in the kernel. UTL lies between the application and the UDP and TCP services provided by the operating system. (Section 3.5.1 highlights the benefits of this approach.) The UTL code is actually linked in with the application itself. All transport layer processing apart from the message framing and demultiplexing provided by UDP—e.g., acknowledgments, retransmission, reordering, duplicate detection, flow-control, and congestion control—is done within the same process as the application. In the current version, cooperative multitasking is used to share the

[31] Some references in the literature use the term "user-space", emphasizing the division of virtual memory between user application memory, and memory reserved for the operating system kernel; see for example (Edwards and Muir, 1995).

CPU between the application and the background transport layer functions (for example, acknowledgment processing, and managing retransmission timeouts) within the application process. Section 3.5.7 discusses the pros and cons of this approach vs. a multiple process or multiple thread implementation, while Section 3.8 surveys related work on user-level protocol implementations.

3.4 Formal specification of rules for composing UTL layers

In this section we provide formal definitions for the rules for composing mechanisms from UTL layers and protocol parameter specifications.

3.4.1 Definition: UTL protocol parameters vector

A *UTL protocol parameters vector* is a *n*-tuple consisting of ordered pairs, (v_i, m_i), ($1 \leq i \leq n$), where:

- *n* is the number of UTL protocol parameters listed in Table 3.3
- Each v_i, ($1 \leq i \leq n$) is either ε, representing the empty value, or it is a specific value that should be used for the *i*th parameter from Table 3.3
- Each m_i, ($1 \leq i \leq n$) is a Boolean value indicating whether the value v_i is mutable. m_i = *true* indicates that the value v_i may be changed, while
  m_i = false indicates that the value may not be changed.

We also use the following notations for UTL protocol parameter specifications:

- *pv.x* indicates the v_i value for the protocol parameter from Table 3.3 named "x"; for example, *pv.maxXmits* is equivalent to the notation v_1.
- *mutable(pv.x)* refers to corresponding m_i value. ❑

3.4.2 Definition: UTL QoS specification

A *UTL QoS specification* is a tuple $Q = \langle o, r, d, c, a, s, e \rangle$,
$Q \in \{O \cup \{\varepsilon\}\} \times \{R \cup \{\varepsilon\}\} \times \{D \cup \{\varepsilon\}\} \times \{C \cup \{\varepsilon\}\} \times \{A \cup \{\varepsilon\}\} \times \{S \cup \{\varepsilon\}\} \times \{E \cup \{\varepsilon\}\}$.

Each element in the tuple representing a UTL QoS specification indicates a value for a specific QoS parameter from Table 3.2, or the null value ε signifying a "don't care" condition. ❑

3.4.3 Definition: Well-formed UTL stack

A *UTL stack specification* is a tuple $s = \langle \boldsymbol{L}, pv \rangle$, where

- $\boldsymbol{L} = (L_0, L_1, L_2 \ldots L_{(n-1)})$ is a list of n UTL layers, and
- pv is a UTL protocol parameters vector. ❑

A *well-formed* UTL stack is defined recursively as follows:

- *Base case:* A UTL stack $s = \langle (b), pv \rangle$ is well-formed, where (b) is a singleton list containing only b, one of the UTL bottom layers listed in Table 3.4, and pv is any UTL parameter list.
- *Recursive step*: Any UTL upper layer u placed on top of a well-formed UTL stack $s = \langle (L_0, L_1, \ldots L_{(n-1)}), pv \rangle$ forms a new well-formed UTL stack
  $s' = \langle (L_0, L_1, \ldots L_{(n-1)}, u), pv \rangle$, provided that the QoS specification $Q = stackQoS(s)$ meets the minimum QoS service requirements of u, as specified in Table 3.5. (see definition of the stackQoS function below.) . ❑

3.4.4 Definition and algorithm for function *stackQoS*

The function *stackQoS* maps a well-formed UTL stack $s = \langle \boldsymbol{L}, pv \rangle$ to a UTL QoS specification $Q = \langle o, r, d, c, a, s, e \rangle$. The following algorithm can be used both to determine whether a stack $s = \langle \boldsymbol{L}, pv \rangle$ is well-formed, and if so, the value of *stackQoS*(*s*):

(1) Compute the QoS specification Q_0 of the stack $s_0 = \langle (L_0,), v \rangle$ from the values and functions in Table 3.4.

(2) For each layer L_i, $1 \leq i < n$:
Lookup the minimum QoS requirements of L_i, in Table 3.5. If Q_{i-1} does not meets or exceeds these requirements, report that *s* is not well-formed and halt. Otherwise, compute the QoS specification Q_i of the stack $s_i = \langle (L_{0,,} \ldots, L_{(i),,} pv) \rangle$ through the values and functions in Table 3.5.

(3) return Q_{n-1} as the value of stackQos(s). ❑

3.5 Design issues

As with any complex software project, the development of UTL involved many design decisions. This section is not intended as an exhaustive list; rather it is an overview of the more important and/or interesting decisions. These design issues include:

- *User-level vs. Kernel-level* development: Should UTL be developed exclusively in the kernel (as TCP and UDP traditionally are), exclusively at user level, or in some hybrid form following the model of previous work on user-level protocol implementations? (Section 3.5.1)

- *Service model:* To provide a variety of transport services through a single API, there needs to be some commonality among these services. What should the common features be? (Section 3.5.2)
  In particular, we discuss two aspects of the common UTL service:

  – Connection-oriented vs. connectionless service (Section 3.5.3)

  - Message-oriented vs. byte-oriented service (Section 3.5.4)

- *Minimizing data copies* for efficiency: How can an extra data copy between the UTL implementation and the application be avoided? (Section 3.5.5)

- *QoS negotiation:* Should UTL include QoS negotiation? Our answer to this question may be somewhat controversial. (Section 3.5.6)

- *CPU scheduling:* Is transport layer background processing (e.g., retransmissions, processing of control packets) handled by a daemon process, a separate thread, or via cooperative multi-tasking? (Section 3.5.7)

- *I/O multiplexing*: How does a process multiplex I/O from UTL file descriptors with I/O from non-UTL file descriptors? (This section motivates the RAW mechanism in UTL.) (Section 3.5.8)

- *Application-Transport flow control:* Should UTL read and write functions be blocking or non-blocking, and how is buffering of data in the transport layer handled? This issue is related to, yet distinct from the issue of flow control between transport layer entities. (Section 3.5.9)

3.5.1 User-level vs. kernel-level development

In most operating systems, transport layer protocols are implemented inside the kernel, as is the case in many popular implementations of Unix. So one might assume that the best place to develop alternative transport layers would be in the kernel. The other alternative is to develop this functionality at user level. A user-level implementation can be a separate daemon process running at user level, a separate thread (or set of threads) within the application, or simply a set of functions called by the application to simulate cooperative multitasking. Once the decision is made to pursue a user-level implementation, the choice from among these three alternatives

(daemon process, threads, or function) is a separate design issue that we consider in Section 3.5.7.

The notion of implementing transport layer functions at user level sometimes causes confusion. The transport layer API often represents both:

- the boundary between the transport and application layers, and
- the boundary between kernel functions and user program functions.

While the first of these is always true by definition, the latter need not be the case: as we note in Sections 3.5.7 and 3.8, many researchers have investigated user-space implementations of transport layer protocols.

There are three main advantages to pursuing a user-level implementation rather than a kernel-based implementation. The first advantage is portability. User level code is far more portable than kernel level code. While there are platform and operating system specific differences among machines, these are far better hidden from the application programmer at user level than from the system programmer who is extending the operating system.

The second advantage is stability. Extensions to the kernel are far more likely to have to be changed when the operating system is upgraded. While user-level code is not immune to problems introduced by OS upgrades, providers of operating systems generally try to shield user-level code from changes to the extent possible, and provide backwards compatibility. Kernel code, by contrast, is considered fair game for overhaul, since it is supposed to be a black box to operating system users and developers.

Third, kernel level development poses a unique set of difficulties. A segmentation fault in a user-level program ends that program, while a segmentation

fault in the kernel may well crash the operating system. Rebooting for each testing/debugging cycle can make kernel development far more time-consuming.

On the other hand, the main disadvantage of implementing a protocol at user level is the potential performance penalty. When a transport entity in implemented in the kernel, it can be interrupt driven, and the CPU scheduling algorithm can give consideration to the real-time nature of packet arrivals and time-out events. By contrast, when the entity implementing the protocol is subject to the context switches at any time, it is more difficult to ensure that protocol events are handled in a timely fashion. Context switches may contribute to burstiness in the transport protocol processing. During a context switch, while one of the cooperating transport entities does not have the CPU (let us call this side A), the other side (side B) may send a large number of TPDUs. When A gets the CPU again, it may end up sending B a large burst of acks, all at once. This burst of acks may cause the flow-control window to open, resulting in another large burst of data from B to A. Thus, a cycle of bursty behavior may result. Such burstiness can cause throughput to drop since the sending side may become blocked waiting for acknowledgments to open the window.

Even with the disadvantages of a user-space implementation, in the end, we felt that the benefits of portability outweighed the performance considerations for our purposes. As part of UTL, we have implemented ordered/reliable services, partially-ordered/partially-reliable services, and unordered/unreliable services, *all* operating at user level as part of the same framework. We postulate that by using this approach, we can make fair comparisons among various kinds of transport service. That is, we believe that our results that can give us a good indication of how those services might perform relative to one another if all were implemented in the kernel.

Performing this comparison first with a user-level implementation can provide an indication of whether the (considerable) effort of developing a kernel-level implementation is worthwhile.

If the answer to this last question turns out to be yes, we note that some parts of UTL are particularly appropriate for migration into the kernel, while others might be better left as a user-level library. One reasonable alternative might be to put the functionality corresponding to the lowest layers of UTL (i.e., TXL, KXP, KX2 and KX3) in the kernel, while retaining the ordering functions represented by layers POL and TOL as a user-level library.

3.5.2 Service model

The primary purpose for which UTL was developed was the comparison of partially-ordered/partially-reliable transport service with ordered/reliable and unordered/unreliable transport service. One approach to this comparison might be to compare POCv2 to TCP and UDP. However, as Table 3.6 illustrates, many differences exist between TCP and UDP other than order and reliability. TCP is connection-oriented and based on the model of a reliable byte-stream (similar to the Unix concept of a file being a stream of bytes). UDP is connectionless and message-oriented—UDP messages are atomic units that are delivered either completely, or not at all.[32] Setting aside POCv2, for the moment—even just to write an application that can run over both TCP and UDP, some common service model is necessary if the special-case code is to be avoided.

[32] The assertion that UDP provides atomic delivery of messages ignores pathological cases where the application provides insufficient buffer space for an incoming datagram; the result in those cases is system dependent, as noted in (Stevens, 1994, p. 160).

Since the focus of our work is order and reliability, we have chosen to bridge the gap between TCP and UDP by basing UTL on a common service model.

Thus, all UTL services are *connection-oriented* and *message-oriented.* The next two sections provide the rationale for this design choice.

3.5.3 Why all UTL services are connection-oriented

Connection-orientated service is necessary for any protocol providing reliability, since to provide reliability, an initial sequence number must be established. While connection establishment is not necessary for an unreliable service, we are more interested in the comparison between partially-reliable and reliable services, both of which require connection-orientation. Moreover, even when comparing against unreliable services, we do not consider the overhead of establishing a connection to be a significant factor in computing statistics such as delay and throughput. Connection establishment overhead is primarily a performance issue for applications that are seeking to carry out a request-response transaction in 2 or 3 TPDUs—for example, DNS queries. By contrast, our experiments will involve a data transfer of tens or hundreds of TPDUs, so making all services connection-oriented does not seem an unfair burden for comparing unreliable services to reliable services. For true request-response applications, order is irrelevant since there is only one TPDU in either direction. Thus a protocol such as VMTP (RFC1045) or T/TCP (RFC 1379, RFC1644) is a more appropriate choice. Even these protocols do not really avoid the establishment of a connection; rather in essence, they avoid repeating a three-way handshake for connection establishment by saving state from previous communications. If one takes the view that a protocol is connectionless if and only if

it is stateless, in some sense even these protocols are not connectionless, but rather a third type of protocol, as argued in (Iren et al., 1999).

3.5.4 Why all UTL services are message-oriented

The choice to make all UTL services message orientated has roots in both theoretical and practical concerns. From a theoretical point of view, the Application Layer Framing principle suggests that the proper unit of transfer for upper layer protocols (Transport and above) should be an Application Data Unit (ADU)—which is to say, a *message*. From a practical point of view, one can argue that the message orientation is providing a service that the application would frequently have to provide anyway, and at minimal overhead.

To see why message-oriented service would have to be provided by the application anyway if it were not done in UTL, first note that since TCP does not mark message boundaries, any application written directly on top of TCP's service has the burden of marking the message boundaries itself. Record boundaries can be identified through

- character counts (a header on each message containing the message length), or
- by including explicit records boundaries, e.g., a special character such as the newline character (this is what most ASCII-based protocols such as FTP and SMTP do in practice.)

Marking these boundaries is not such a problem for the sender. The larger burden falls to the receiving application. If the receiving application issues a read request to TCP for up to *n* bytes, it must be prepared to accept any number of bytes between 1 and *n*. There is no way for an application to say to TCP, “Give me the next message in its entirely, and only the next message”. Rather, if the receiving

application wants to process an entire message as a unit, the application must do its own buffering of the bytes of the message until all are delivered. If on any given read request, less than n bytes are delivered, the application must repeatedly loop, requesting additional bytes until the entire message arrives.

The message orientation of UTL becomes a fairness issue only when making claims of comparison vs. TCP. The TX service of UTL provides a message-oriented service on top of TCP's byte-stream service. The TX service accomplishes this by having the sender prepend a four-byte length field to each message. The loop to build up the complete message at the receiver is then placed inside UTL, and the message is given to the application only when complete. Putting the message boundary marking function inside UTL relieves the developer from having to write the code to perform this function. However, it may also introduce an extra delay, an extra price paid for message-oriented service.

The issue of fairness hinges on whether the application can make use of individual bytes of a message as they arrive, or whether an entire message must be received before processing can begin. For example, suppose a GIF or JPEG decoder is written in such a way that it can accept as little as one byte at a time, and keep all of its state between calls. In this case, it is conceivable that an application for displaying images might perform better over a byte-stream service than a message-oriented service. An experiment to investigate the penalty of message service vs. byte stream service is suggested in the future work section of this dissertation; in the meantime, one should keep this limitation in mind when interpreting results based on UTL's TX service.

For other comparisons however—that is, comparisons among UDP-based services, which includes all the performance experiments presented in this dissertation,—we claim that making all UTL services message-oriented is a reasonable choice, supported by the Application Layer Framing principle.

3.5.5 Minimizing data copies for faster throughput

A major theme in work on improving the throughput of protocols (especially in the related work on user-level protocol implementations) is the issue of minimizing data copies (Clark and Tennenhouse, 1990; Edwards and Muir, 1995, Thekkath et al**.,** 1993). Assuming a single processor architecture, if that architecture can copy data at *s* bits/second, then *s* represents an upper-bound on the throughput of any transport protocol. This upper bound is a consequence of the fact that, at the very least, data must be copied from the network into memory. If there are *k* data copies, then the maximum throughput drops to *s/k*. Thus, it is advantageous to have has few data copies as possible.

In a typical Unix architecture, a minimum of two data copies takes place for each incoming TPDU. The first is from the network interface card into kernel memory. In the case of Berkeley-derived TCP/IP implementations, arriving data link PDUs are placed into in-kernel memory buffers called *mbufs*. Mbufs are designed with pointers so that they can be easily manipulated for adding and stripping off headers and trailers for encapsulation/de-encapsulation of PDUs. Mbufs are also used for other aspects of the operating system as well, and as such, are a protected resource. Therefore, when the application reads data from a TCP or UDP socket, the data must be copied a second time, from the in-kernel memory into user space.

Table 3.6 Services provided by various transport layers

		API	Berkeley Sockets		POCv2	UTL Common Service	Example UTL Services					
		Transport Service	TCP	UDP			PO3	TX	UC	XP	XP3	SP3
Service Parameters		Connection Orientation	CO	CL	CO	CO	CO					
		Service Orientation	Byte	Mesg	Mesg	Mesg	Mesg					
		Order	O	U	PO	—	PO	O	U	U	U	O
		Reliability	R	U	PR	—	R	R	U	PR	PR	R
	Flow Control	Transport-Transport	Y	N	Y	—	Y	Y	N	Y	Y	Y
		App-transport at sender	Y	N	Y	—	Y	Y	N	Y	Y	Y
		App-transport at receiver	Y	N	Y	—	Y	Y	N	N	Y	Y
		TCP-friendly congestion control	Y	N	Y	—	Y	Y	N	N	Y	Y

Legend:

CO: connection-oriented CL: connectionless
Byte: byte-stream Mesg: message-oriented
O: Ordered PO: Partially Ordered U: Unordered
R: Reliable PR: Partially Reliable U: Unreliable

Since UTL is built on top of UDP and/or TCP, these two data copies are present. To preserve the sockets API exactly, it would have been necessary to add a third data copy. Instead, the UTL API was designed in such a way as to avoid any extra data copies.

In the sockets API, the `read` call takes three parameters:

```
int read(  int fd,  /* file descriptor /
   char buf /* pointer to buffer */
   int len /* length of buffer */);
```

The pointer `buf` points to some static or dynamically allocated storage in the application's memory where the data should be placed, and `len` points to the maximum amount of data to be read. The return value is the amount of data *actually* read, which may be less than the value `len`.

By contrast, the `utl_Read()` call takes two parameters:

```
int utl_Read(   int fd,  /* file descriptor /
   char **msg_H /* handle of packet /);
```

Here, the parameter `msg_H` is a handle; that is, a pointer to a pointer. The application supplies the address of a (`char *`) variable where a pointer to the incoming TSDU should be stored. The return value of `utl_Read()` is the length of the TSDU that is now pointed to by `*msg_H`. After a successful `utl_Read()` call, the application now has *custody* of the memory at this pointer—that is, it may use that memory for whatever purpose it wishes, but when it is done, it has the responsibility to free that memory via a call to the function `utl_FreeFrame()`. The fact that the `utl_Read()`call returns a pointer to memory already allocated by UTL avoids an extra data copy at the expense of a deviation from the standard Berkeley Sockets API/ If `utl_Read()` followed the Berkeley Sockets API strictly, an extra data copy would

be required to move data that UTL had already read from UDP or TCP into the application's data space.

A similar procedure is used for output; when an application wants to build a message to transmit over the network, it must first call a UTL function to allocate a special place in memory. UTL automatically prepends sufficient memory for packet headers based on the transport service that has been selected. When the `utl_Write()` function is performed, no data copy is done; only a pointer value is passed. UTL then takes custody of the frame. When the data is actually written to the network, because the space for the headers was already allocated contiguous with the data, the entire TPDU can be written to TCP or UDP in one operation.

3.5.6 QoS negotiation

QoS negotiation is a feature by which the application can specify specific quality of service parameters, and make the establishment of a connection contingent on a minimum service guarantee from the transport layer, for example, ordered, reliable service. In Section 3.3.4, we pointed out that UTL does not currently have negotiation. If applications built over UTL were to be used in a production environment, a QoS negotiation feature would be essential, thus future work on UTL might include the addition of such a feature. For our current research purposes of experimenting with flexible transport QoS tradeoffs, we claim that a negotiation facility would only get in the way.

To understand this point, imagine how the interaction between the application and UTL might proceed if such a facility were in place. If UTL provided QoS negotiation, at connection establishment time the connection request would include a specification of the requirements in terms of order, reliability,

synchronization, etc., either in lieu or, possibly in addition to a request for a specific mechanism. Then, if the mechanism could not meet the service requirements of the application, the connection would be rejected. While rejecting a connection request when the service cannot meet the applications needs is clearly useful for a production application, this rejection would have no additional benefit in the context of evaluating application performance over various transport services. The results of an experiment that rejects a connection request are not interesting from the standpoint of delay, throughput or jitter. On the other hand, the results of an experiment where the mechanism does *not* precisely match the QoS needs of the application can be interesting.

For example, suppose a particular application requires partial order, but not total order, and consider what will happen in two cases:

(1) The mechanism used provides total order service (e.g., SP2)

(2) The mechanism used provides unordered service (e.g., X2)

In the first case, provided that the sender uses a linear extension of the partial order as the original sending order, the application will perform correctly, albeit possibly with worse performance. We claim the application will perform correctly, because the sending order will be preserved, and is a legal linear extension of the PO. On the other hand, there will be performance degradation, at least at some level of network loss, if our central hypothesis holds true: that partially-ordered transport service provides performance improvements over lossy networks.

In the second case, the application may fail to perform correctly if a packet reordering occurs, since the transport layer will not detect this. However, for purposes

of transport protocol experimentation, it is still useful to allow the connection to proceed in spite of potential failures. It allows one:

(1) to demonstrate the failure mode that occurs when an inappropriate transport service is used; (for example, to show what a ReMDoR document looks like when explicit release synchronization is *not* provided)

(2) to demonstrate that at *low* loss rates, failure does *not* occur (because there is no reordering), and

(3) to experimentally estimate the failure probability as a function of the loss rate.

ReMDoR provides another example. While complex multimedia documents such as the `paris.pmsl` and `military.pmsl` (see appendix) require partial order and explicit release synchronization, the simple single image documents used in the NETCICATS experiments (Iren, 1999b) require neither of these. On the other hand, they *do* require total order in the case of experiments with GIF files. UTL provides the experimenter with flexibility by allowing the selection of *any protocol with any parameter for any application.*

Thus a QoS architecture that prevents mismatches between application needs and the QoS the transport layer provides, while certainly necessary in a production environment, might just interfere with the performance experiments for which UTL was initially designed.

If programs based on UTL reach a point where they may be useful as production applications, a QoS negotiation facility should be added to UTL. This facility would allow an application to be agnostic concerning UTL mechanisms, and

instead, to specify quality of service by indicating a list of minimum service requirements for each QoS parameter that would ensure correct operation. UTL would then search its list of mechanisms, make an appropriate choice if possible, and establish the connection. If UTL determined that more than one mechanism could meet the requested quality of service, it could either

(1) make the best choice, according to some heuristic,

(2) make an arbitrary choice if no useful distinction can be made, or

(3) provide a list of appropriate mechanisms to the application

The third choice would allow the application to present the list of acceptable mechanisms to the user. The user could then make a selection from among only the mechanisms guaranteed to perform correctly according to the minimum QoS needs of the application. Of course, to retain the capability for experimenting with failure modes resulting from an inappropriate selection of transport service, there should be some way of disabling any such QoS negotiation mechanism.

3.5.7 CPU scheduling in UTL via cooperative multitasking

UTL is an example of a *user-level* (sometimes called *user-space*) protocol implementation. This section discusses a key challenge facing the developer of a user-level protocol implementation, that of CPU scheduling. We first review the handling of CPU scheduling for transport processing in the BSD Unix kernel, since this serves as the reference implementation of TCP/IP, and the yardstick against which previous researchers in this area measure their designs. We then highlight two problems that arise in CPU scheduling for user-level transport protocol implementations: the *top-half/bottom-half* problem, and the *lingering connection*

problem. We first define these problems, then survey the solutions to these problems described in previous work, and finally explain how the solutions in UTL differ from these.

The BSD Unix kernel (McKusick et al., 1996) serves as the reference implementation for the Internet protocol suite, which is arguably the most widely used protocol suite in the history of computing. In BSD Unix, transport protocol processing takes place in the kernel. Requests from the user trap to system calls processed synchronously in the so-called *top-half* of the kernel. Timeouts and incoming packets are processed via interrupts in the so-called *bottom-half* of the kernel. Thus, the normal CPU scheduling mechanisms of Unix handle the scheduling of all transport layer processing. Since all of this processing takes place inside the kernel, sharing data between the top-half and bottom-half routines is not difficult.

Research on user-level implementations of TCP (Thekkath et al., 1993; Edwards and Muir, 1995) highlights two key design problems, which we have given the following names:

- The ***top-half/bottom-half division*** problem: How are the top-half and bottom-half processing to be scheduled in a way that they do not interfere with one another, or the application? Avoidance of context switching and sharing of data are two concerns.

- The ***lingering connection*** problem: The semantics of the Berkeley Sockets API allow TCP connections to live longer than one side of the application process. Specifically, an application sending data such as a server, may write the last window's worth of data to the transport layer, and then close the connection. The Berkeley Sockets `close()` operation is by default non-blocking[33], so it returns immediately and allows the application process to terminate. However, the kernel is still delivering data on behalf of

[33] The SO_LINGER option can be used to make it blocking.

> the terminated process. If retransmission is handled by the application, either the `close()` operation must block until all data has been transmitted, or the application must hand-off this responsibility to some other process.

If the transport layer processing is to linked in with the application, how is this processing to be scheduled? Here are three possible solutions

Multiple processes

In the first solution, the application process performs the top-half processing when the API functions are called. The bottom-half processing is performed by a separate daemon process; either (a) a single process shared among all processes on the host, (b) a single process per running application, or (c) a single process per connection. Since there are two separate processes, the top and bottom-half are scheduled independently by the operating system. Processing for lingering connections can be handed off to the separate daemon process.

Examples of this approach include (Thekkath et al., 1993) which describes an implementation of option (a), and (Edwards and Muir, 1995) which describes an implementation of option (b). This approach has the advantage that it allows the implementation to closely model the BSD reference implementation.

It also has two main disadvantages: context switching between the top-half and bottom-half code, and the complexity of sharing data between the top and bottom-half processes by message passing or shared memory with mutual exclusion. In the conclusions section, (Edwards and Muir, 1995) notes that

> A consequence of [the multi-process architecture was that] our TCP was constantly context-switching, which lowered performance. Also, the multi-process nature of the implementation meant that we seemed to spend as much time worrying about concurrency as about our real goals.

Multiple threads

An alternative to the heavyweight context switching of the multiple process solution is use lightweight threads, for example, POSIX threads (Nichols et al., 1996). In this solution, the top and bottom-half processing are implemented by different threads. However, while one no longer has to deal with the difficulties of message passing or shared memory, the complexity of enforcing concurrency and mutual exclusion remain an issue. Furthermore, as (Edwards and Muir, 1995) note, multiple threads do not address the lingering connection problem, since all threads die with the application.

However the paramount concern for our project was whether POSIX threads and X-Windows could co-exist in the same application; at the time we were undertaking our initial design in Fall of 1995, it was unclear whether Xlib was thread-safe. Since our ReMDoR browser was an X11 application, this concern led us to avoid the multiple thread model. At this point, we are more confident the X11 and POSIX threads can be used together, and we are therefore considering this approach for a future redesign of UTL.

Signal Handlers

Another option is to have a single process perform all processing, and use the SIGALRM signal with signal handlers to implement the bottom-half processing. (Edwards and Muir, 1995) points out that since the application may already be making use of the signals, it would be necessary to "silently take control of alarm handling" through wrapper functions. While this approach is feasible, we again avoided it due to concerns about compatibility with X11; the X Consortium's official recommendation

is that X11 applications should avoid signal handling, advice that is reinforced by (Heller and Ferguson, 1994).

Cooperative multitasking

This leads us to the solution used in UTL: cooperative multitasking. Our approach builds on an observation in (Edwards and Muir, 1995) in their suggestions for future work section:

> One possible approach is to exploit the fact that most application programs use sockets in 'blocking mode', sleeping on the socket till there is more data to receive or space available for sending. If a packet arrives during this 'sleep', we could perform receive processing in the context of the application process, which would eliminate context switches in the common case. With these improvements, even better performance should be achievable.

While Edwards and Muir observation is correct for many applications, it overlooks two important (at least to us!) classes of applications:

(1) **Applications that block on the `select()` call.** Applications such as the ReMDoR server, may need to process more than one I/O stream simultaneously: for example, a socket listening for new connection requests, and several output sockets for connections being actively served. These applications block not on a `read()` or `write()` operation, as suggested by Edwards and Muir, but rather on a `select()` system call.[34]

(2) **Applications that use callbacks for network I/O (e.g., X11)** X-Windows applications such as the ReMDoR browser must use callback functions to process I/O streams with waiting data. This is because the X11 library has its own event loop, which implements a wrapper around the `select()` system call.

---

[34] The select() system call allows an application to register interest in a set of file descriptors, and be notified when *any* of them become readable or writeable, or when a timeout is reached. The timeout is passed by reference; a timeout of zero indicates a poll, while a timeout pointer of NULL indicates a blocking call. (Stevens, 1998)

> Therefore, in an application using the X11 protocol, to read (or write) data from the network, the programmer must provide a callback function that gets called automatically by X whenever the socket becomes readable (or writeable).

Therefore, our approach follows Edwards and Muir's suggestion, but in a way that considers not only applications that block on `read()` or `write()`, but also provides for applications that block on `select()`, applications that use callbacks, and applications that may wish to avoid blocking on network conditions altogether.

In UTL, all protocol processing is done in the application, and all data structures for transport layer code are in user space. Because we build most transport functionality over UDP, we rely on UDP to provide assignment of port numbers for incoming packet demultiplexing, a function that both (Thekkath et al., 1993) and (Edwards and Muir, 1995) implemented with a single trusted connection server process per host.

Top-half processing is done synchronously when the application calls the UTL API functions `utl_Listen()`, `utl_Connect()`, `utl_Accept()`, `utl_Write()`, `utl_Read()`, and so-forth. The bottom-half processing, including timer management and processing of incoming packets, is done in a function called `utl_IO()`. This `utl_IO()` function is the heart of the UTL implementation, for it is responsible for timer management, moving data up and down through the layers of the UTL protocols stacks for each mechanism.

Following Edwards and Muir's suggestion, each time one of the API functions is called, it in turn calls the `utl_IO()` function to handle any pending bottom-half processing. If the API function is blocking, for example, a `read()` call when no data is available, the application will remain in the utl_IO function until data becomes available. To avoid busy-waiting, utl_IO first handles any processing for

timers that have expired, then blocks on the `select()` system call waiting for either input from the network, or the expiration of the next timer.

In addition, there is a `utl_Service()` call, that emulates the semantics of a `select()` system call, and provides a direct means to invoke the `utl_IO()` bottom-half processing. An application that is designed to block on the `select()` system call can simply block on `utl_Service()`.

Finally, in the case of the XWindows application, the callback function for the socket file descriptor can include a call to `utl_Service()`. In addition, XWindows allows a callback for background processing (a function that is called when all other processing is idle). The `utl_Service()` function can be provided to this callback as well.

Applications that do not wish to block on `read()` or `write()` operations can simply make a non-blocking `select()` call (by passing a delay value of zero) to check whether a given `read()` or `write()` operations would block before attempting it.

The advantage of our approach is its simplicity; there are no mutual exclusion problems since the application has only one thread of control. A disadvantage is that the application programmer has a particular burden: specifically, to structure the code in such a way that `utl_IO()` is called frequently enough to guarantee that background processing of acks and timeouts takes place as needed. We refer to our approach as “cooperative-multitasking” because the application and the transport layer must cooperate to share the CPU; specifically, the application must periodically yield its share of the CPU time to the bottom-half transport layer processing. If `utl_IO()` is not called often enough, performance will suffer, and if

`utl_IO()` is not called at all, the protocol will deadlock. By embedding bottom-half processing in the calls to top-half functions, we have made it likely that even without any particular planning on the part of the application programmer, `utl_IO()` will get called often enough for the typical network applications paradigm—a read and process loop that blocks on `read()`, `write() or select()`, and then performs a modest amount of processing[35] on the next piece incoming or outgoing data. However, there may be some applications where this arrangement is unsuitable—for example, applications where there is occasionally a requirement to do a long computation in between network operations. In such cases, the application programmer must schedule occasional calls to `utl_Service`() to ensure that the appropriate transport layer processing takes place.

To free the application programmer from the burden of having to consider CPU scheduling at all, we plan to investigate a multiple thread approach as an alternative for future implementations of UTL. Although the use of multiple threads introduces the difficulties of mutual exclusion for the UTL implementer, we argue that making the bottom-half CPU scheduling transparent to the application programmer would make UTL a more useful and robust tool, and would therefore be worth the effort.

Another disadvantage of UTL is that we offer no solution to the lingering connection problem except the following: when an application is terminating, it must call the function `utl_Finish()`. This function blocks until all pending connections are complete. This approach to the lingering connection problem allows the `close()` call to be non-blocking, but it does not allow the server process to complete until all

[35] Anywhere up to around 200ms or so.

processing is finished. As Edwards and Muir point out, this problem would afflict a future thread-based implementation as well. However, while the lingering connection problem has been highlighted as a crucial issued by both Edwards and Muir as well as Thekkath et al., and both projects invested considerable effort in addressing it, by contrast, we do not consider this to be a serious problem in practice. We argue that it is probably good for most applications to linger until they receive positive notification that all connections have completed safely and reliably (unless they choose to abort rather than request a graceful close.)

3.5.8 I/O multiplexing, and the need for a RAW mechanism

The `utl_Service()` call described in Section 3.5.7 normally assumes that all file descriptors passed to it represent sockets on which there is an active UTL connection. To provide the semantics of the `select()` system call for a mixture of UTL and non-UTL file descriptors, UTL provides a special mechanism called "RAW". For example, a chat application that must read from both the keyboard and the network can register the keyboard as a UTL connection using the RAW mechanism. UTL will then pass that file descriptor through untouched to the underlying `select()` call in `utl_IO()`, and report the results along with those of the UTL connections.

The existence of the RAW mechanism is the one reason we use the term *mechanism* rather than *protocol* or *service*. Another reason is that the original design for UTL also called for a mechanism (never implemented) for testing, by which information could be read from a local disk file using the UTL API, rather than from a remote host; in this mechanism `write()` operations would be discarded.

3.5.9 Application-Transport flow control

Traditional presentations of flow control in networking textbooks focus on sliding window flow control between two data-link-layer entities or two transport layer entities (Tanenbaum, 1996; Stallings, 1998). However, there is another important aspect to end-to-end flow control frequently overlooked in these discussions. The presentations of the classic algorithms (stop-and-wait, go-back-N, selective repeat) often make the simplifying assumption either that (a) there is an infinite queue between the service provider (e.g., the data link layer, or the transport layer) and the service user (e.g., the network layer, or the application layer,) or (b) that the service user is always immediately available to the service provider to provide a new SDU to send, or to consume an SDU that has been received.

In practice, particularly at the transport layer, such assumptions are unrealistic. Therefore, in addition to the usual window flow control between the transport entities, it is also necessary to provide flow control at the TSAP, which we call *application-transport flow control*. With application-transport flow control in place, if the receiving application stops reading data and deliverable data queues up, filling the transport receiver's buffers, the flow control algorithm will halt the submission of new data at the sending application. TCP provides application-transport flow control through *window advertisements* in each TPDU, and by blocking the submission of new data at the TSAP if there is no remaining space in the window.

In the remainder of this section, we describe two different forms of application-transport flow control:

- *Sender and receiver* application-transport flow control, and
- *Sender-only* application-transport flow control,

and then distinguish between *mandatory* application-transport flow control (as in TCP) and *advisory* application-transport flow control (as in UTL).

Sender and receiver application-transport flow control

The use of sender and receiver application-transport flow control (as in TCP) implies that fixed size buffers are used at both:

- the transport sender (for unsent data and unacked TPDUs), and
- the transport receiver (for data that is undeliverable, or deliverable but still unread by the receiving application)

If the receiver's buffers are full, the window flow control scheme throttles the sender. If the sender's buffers are full, the sender prevents the sending application from submitting additional TPDUs until window space opens up. The net effect is that if the receiving application stops reading data and deliverable data queues up filling the transport receiver's buffers, the flow control algorithm will halt the submission of new data at the sending application.

Both TCP and the KX3 layer of UTL provide sender and receiver application-transport flow control by blocking the submission of new data at the TSAP if there is no remaining space in the window. In TCP, the receiver informs the sender as the remaining window space through two fields in the acknowledgment TPDU: the cumulative ack, and the window advertisement. Because TCP is reliable and ordered, these two values precisely determine the starting and ending byte of the current legal sending window.

In KX3, the fact that the service is unordered and partially reliable makes a scheme based on a cumulative acks and windows advertisements of available buffer

space unfeasible. Instead a novel approach is used which uses the cumulative ack field along with two additional sequence numbers.

(1) A *receiver-least-undelivered* field is sent from receiver to sender to indicate the lowest numbered packet that has not yet been delivered or declared lost. This value represents the lowest numbered packet for which buffer space needs to be reserved.

(2) A *sender-left-edge* field is sent from sender to receiver, indicating the left edge of the sending window; the receiver uses this value to detect when unreliable or partially-reliable messages have been declared lost at the sender.

***Sender-only* application-transport flow control.**

Sender-only application-transport flow control implies that the service provider has a fixed size sending buffer for unsent or unacknowledged TPDUs, but the receiver has an infinite (or more, precisely, an arbitrarily large) queue between the transport receiver and the user application. KXP and KX2 provide sender only application-transport flow control.

Application-transport flow control is mandatory in TCP, advisory in UTL

One additional detail is that in UTL, application-transport flow control at the sender is advisory, not mandatory. In fact, early versions of UTL provided no application-transport flow control at all. An arbitrarily large queue between the UTL user and the UTL transport service is still provided for applications for which this is a desirable feature. Our experience is that to make accurate measurements of delay, application-transport flow control is necessary. A UTL user desiring application-transport flow control can inquire before every `Write()` operation as to whether the `Write()` would result in queuing of data. If it would, the application can choose to

perform other processing, or ask to be blocked in the transport layer service routine[36], until buffer space becomes available.

3.6 Selected service and protocol details for the KXP, KX2 and KX3 layers

In this section, we summarize a few details of the services and protocols implemented by the KXP, KX2 and KX3 layers described in Section 3.3.7. This section is not intended as a complete service/protocol specification, but rather a reference to help the reader better understand the UTL framework and interpret the results of performance experiments involving these protocols.

KXP and KX2 have been extensively tested, while KX3 has received comparatively less testing. For that reason, most of the empirical results reported in this dissertation are based on KX2.

3.6.1 Unordered, *k*-xmit reliable service

All three KXx protocols provide unordered, *k*-xmit reliable service, which is defined in (Marasli, 1999b) as follows:

> A packet with *k*-xmit reliability can be transmitted (original plus retransmissions at most *k* times. If [the] transport sender is still waiting for the ack of a packet after the k^{th} transmission timeout, the packet will be released from [the] transport sender's buffers. Releasing a packet from the sender's buffers without receiving an ack for it is called "declaring that packet lost at [the] transport sender".

[36] `utl_Service()`, discussed in Section 3.5.7.

3.6.2 Flow control

All three KXx protocols provide sliding window flow control between transport entities, and application-transport flow control at the sender. Only KX3 provides application-transport flow control at the receiver.

3.6.3 Packet types

Each of the KXx protocols provides four packet types: SYN, FIN, RDATA and ACK. The SYN (synchronize) and FIN (finish) packet types are used for connection establishment via three-way handshake, and connection teardown via dual half-close exactly as in TCP (RFC793). The RDATA (reliable data) packet type is used to send data, while the ACK (acknowledgment) packet type provides *selective* positive acknowledgments (acks) of SYN, FIN and RDATA packets. All RDATA packets also carry a piggybacked ack field; although the use of this field varies between KXP, KX2 and KX3 (as explained in Section 3.6.4).

3.6.4 Acknowledgments

All three KXx protocols use positive selective acknowledgments (acks). Each TPDU is acknowledged by an explicit ack sent as a separate control packet. In addition, KXP and KX2 have a *lastAck* field in every TPDU, which repeats the sequence number of the most recently acknowledged packet. This extra redundancy in acks helps ameliorate the effect of ack losses, which Marasli's analytic model showed have an important influence on the performance of partially-reliable transport protocols (Marasli, 1999b). KX3, by contrast, supplements the lastAck field with a cumulative ack based on the linear order of sequence numbers. While delivery is still unordered, using this cumulative ack allows KX3 to implement the fast-retransmit and

recovery algorithms of TCP (RFC2581), which preliminary results show can improve delay and throughput.

3.6.5 Sequence numbers

KXP uses a complicated two part sequence number scheme explained in (Golden, 1998) that is tied to the internal management of buffers in the transport sender. This scheme makes analysis and debugging of the protocol more difficult. KXP also lacks congestion control features. In KX2 and KX3, the sequence number scheme is simplified: each successive TSDU submitted by successive `Write()` operations is assigned a consecutive sequence number. This simplification was motivated by a desire to simplify the protocol and was also necessary to implement cumulative acks for fast-retransmit and recovery.

3.6.6 Congestion control

KXP provides no congestion control features whatsoever. Protocols lacking congestion control are considered harmful for use on the wide-area Internet because they unfairly compete for bandwidth (Floyd and Fall, 1999). KX2 and KX3 represent incremental steps towards the goal of a fully *TCP-friendly* implementation of KXP. A TCP-friendly application is defined by Floyd and Fall as one whose "arrival rate does not exceed the arrival rate of a conformant TCP under the same circumstances." Research modeling the behavior of the TCP congestion control algorithms has characterized TCP-friendliness as an arrival rate that is at most some constant over the square root of the packet loss rate. (Floyd, 1991; Lakshman and Madhow, 1997; Mathis et al., 1997.)

Our approach to TCP-friendliness is to emulate TCP's exponential backoff of retransmissions, and the four congestion control algorithms documented in RFC2581: slow start, congestion avoidance, fast-retransmit and fast-recovery.

KX2 always performs exponential backoff of retransmissions, and, if the enableCongestionControl parameter of UTL is set to true, KX2 also enables slow start and congestion avoidance. However, KX2 lacks the ability to implement fast-retransmit and recovery due to its lack of cumulative acks.

KX3 adds fast-retransmit and recovery to KX2, providing what should be a complete emulation of TCP-friendliness. Establishing KX3's TCP-friendliness by measuring its throughput against the ($k / \sqrt{p}$) formula is part of the ongoing work on KX3.

3.6.7 RTO calculation

To calculate the retransmission timeout (RTO), all three KXx protocols use a set of common UTL functions[37] that implement TCP's algorithms for RTO calculation—Karn's algorithm (Karn and Partridge, 1987) and Jacobson's algorithm (Jacobson, 1988). In the process of implementing these algorithms, we made a counter-intuitive discovery. The reference implementation of TCP uses a coarse granularity for the RTO timer (500ms). We assumed that more accuracy would be better. On the contrary: it turns out that improving the accuracy of this timer can have negative effects! The validity of Jacobson's formulas is predicated on a coarse measurement of the RTT. In this section, we explain the performance problem that can result if these formulas are applied naively to overly accurate measurements of RTT. To highlight this counter-intuitive result, we briefly review the main ideas of

[37] Programmer's note: These common functions are in the `utlTool` module.

the Jacobson algorithm, explain why it fails if accurate timers are used, and explain how we addressed this in UTL.

If RTT were a fixed value, the ideal value for RTO would be RTT+t_{safe}, where t_{safe} is the time necessary to process an incoming acknowledgment and thus prevent an unnecessary retransmission. The value RTT+t_{safe} allows for a retransmission at the earliest possible instant the sender can detect with certainty that a failure occurred in either the transmission or the acknowledgment. Unfortunately, at the transport layer, considerable variation in RTT is caused by queuing delays, routing changes, and context switching in the end hosts. Therefore, the mean and variance RTT must be estimated, and the t_{safe} value must include both the minimum processing time as well as allowing for any variance.

Jacobson's algorithm (Jacobson, 1988; Stevens, 1994) computes RTO by sampling the RTT values, and using the samples to estimate the average, *A*, and mean deviation, *D*, of the RTT via exponentially decay. If *M* is the measured RTT sample:

$g = 0.125$ $\qquad A_{new} = (1-g)A_{\text{old}} + gM$

$h=0.25$ $\qquad D_{new} = (1-h)\text{D}_{\text{old}} + h(|M-A_{\text{old}}| - D_{\text{old}})$

RTO is then calculated via RTO=(A + 4D). The intuition behind this formula is that four times the deviation in RTT samples provides enough t_{safe} time to prevent premature retransmissions.

However, in his explanation of this calculation, Jacobson assumes that the values are measured in so-called "ticks", as in the reference BSD Unix implementation of TCP. Table 3.7 shows the tick size used for the estimation of A and D in BSD Unix TCP vs. UTL.

Table 3.7 Tick sizes for RTT/RTO calculation in TCP and UTL

	UTL	BSD Unix TCP
RTO	1ms	500ms ($^1/_2$ sec)
Mean RTT (A)	1ms	62.5ms ($^1/_{16}$ sec)
Mean Deviation (D)	1ms	125ms ($^1/_2$ sec)

The end result is that TCP's RTO ends up with what (Jacobson and Karels, 1998) calls a bias of 1.5 to 1.75 ticks[38] That is, the RTO ends up being, on average, somewhere between 750 to 825 milliseconds higher than the true value of A + 4D. In practice, this bias is usually sufficient to ensure that if D goes to zero, TCP still does not timeout prematurely.

In early versions of UTL by contrast, the tick value was 1ms. This accuracy led to a pathology in which certain connections would experience catastrophic drops in throughput after a few hundred packets. Investigation of this phenomenon showed that it was due to premature retransmissions due to an inadequate RTO value. Where the bias in RTO for TCP is between 750 to 825ms, the equivalent bias in early UTL versions was on the order of 1 to 2ms. As a result, if during some interval, the RTT remained stable long enough, the D factor could drop to a small value, e.g., between 10-15 ms. If the RTT then suddenly increased by a value greater than 4D, a premature retransmission resulted. This sequence of events led to a syndrome where *every* packet was transmitted twice: because of Karn's algorithm[39],

---

[38] 1.5 ticks was in the printed SIGCOMM '88 proceedings; a "revised 1992 version of the paper" on their web site says 1.75.

[39] RTT measurements are not used for any packet that is retransmitted, because the value is ambiguous; there is no way to know whether an acknowledgment comes from the initial transmission or the retransmission. (Karn and Partridge, 1987)

UTL could not update the RTT again unless and until the true RTT dropped below the RTO, enabling UTL to update the RTT estimates.

Our solution to this problem was to explicitly introduce an extra 750ms safety margin into the RTO calculation: that is, we use A + 4D + 750ms as the RTO in UTL. This value is, on the one hand, arbitrary: one can describe specific scenarios where it would not be sufficient to prevent the syndrome from occurring. On the other hand, this value is based on emulating the behavior of a successful protocol (TCP) and seems to have eliminated the problem in practice.

3.7 UTL development, testing and debugging

This section describes the development, testing and debugging tools and processes used for development, testing, debugging and enhancement of UTL. We make no claim that these techniques are particularly novel. Nevertheless we include them because (1) a brief explanation of these tools and processes may help the reader appreciate the scope and usefulness of UTL, and (2) other developers of user-space protocol libraries may find this discussion of practical issues helpful.

3.7.1 UTL development

As with most large pieces of software, UTL is not the work of a single person. However, the author of this dissertation has been the chief designer and architect of UTL, has supervised its construction, and has written at least half of the UTL code. Significant additional contributions have been made by Edward Golden (Golden, 1997), and Mason Taube.

While the need has been recognized to move to a formal version control system such as CVS[40], that has not been done with UTL to date. Instead, an informal system for managing versions has evolved into a set of standard procedures that have been followed since release 0.74 (Nov. 1997). A master directory is maintained for the UTL source code, which is updated only when a new production release is made. New production releases are made only after extensive testing, and only when there is a significant change (either a bug fix, or an enhancement) that is useful for our protocol research. To provide a general idea of the process of development, Table 3.7 shows the numbers and dates of production releases done since version 0.74, and descriptions of some of the key changes in the recent releases.

3.7.2 UTL testing

To test the functionality of UTL, at first a series of ad-hoc test programs was used. Later it became clear that a more comprehensive approach was needed to provide assurance that new releases did not introduce new bugs; in the software development community, this is called *regression testing*. For this purpose, a pair of programs called `diag` (for diagnostic) and `rd` (for read) were developed originally by Mason Taube, and later extensively modified by the author. These programs have been used for regression testing in every version of UTL since version 0.80. The purpose of these two programs is to give UTL an extensive workout. No claim is made that these two programs provide any formal test coverage (for example, executing every transition of the Finite State Automata of each layer.) However, they

[40] Concurrent Versions System (ftp://prep.ai.mit.edu/pub/gnu/cvs-1.3.tar.gz).

have been helpful in finding bugs and in providing some level of confidence in a each new release.

The `diag` and `rd` programs operate in three phases as follows:

Phase 1:In this phase, rd operates as a server, and diag operates as a client. For each UTL service to be tested listens for a connection. diag makes the connection for each UTL service in turn, sending exactly one packet before tearing down the connection.

Phase 2: The programs now switch roles, with `diag` operating as a server, and `rd` as the client. For each service, the `diag` program does a listen call, and accepts a single connection from `rd`. `rd` then sends (by default) 20000 messages of 1024 byte each, using the selected protocol, and then closes the connection.

Phase 3: The programs switch roles for a third time. This time, rd listens for each selected service simultaneously, and diag establishes n simultaneous connections, where n is the number of UTL services being tested. The connections are established in random order. Then diag sends 20,000 messages of 1024 bytes each, each time making a random selection from among the n open connections. (This tests for any unexpected interactions among mechanisms, or stray pointers that might corrupt the memory of another layer.) Finally, diag closes each of the n connections, and terminates.

By default, `diag` and `rd` check all defined mechanisms within UTL, and send 20,000 messages of size 1024 bytes for each test. *Every production release since v0.80 has passed this test before being added to the master directory.*

The `diag` and `rd` programs can also accept command line options to test only particular subsets of services, and to change the number and size of test messages.

3.7.3 Debugging macros

Two specific techniques were used in debugging UTL that may be of general interest. These include a set of debugging macros for controlling diagnostic output (described in this section) and a set of wrapper functions for `malloc` and `free` that help to find pointer related bugs (described in the next section).

While general purpose debuggers such as `dbx`[41] or `gdb`[42] have their place, it is sometimes more helpful to put trace output into a program. However, trace output can create certain problems. Network protocols can be considered real-time systems, in the sense that the progress of the computation is governed by events that happen in real time, such as packet arrivals and timeouts. The time it takes to produce diagnostic output may introduce artifacts into the timing of events such that a particular bug does not occur.

What can help is to allow fine control over the level of diagnostic output. If the content and amount of diagnostic output can be controlled, by repeated experimentation, the developer can find the level of output at which the error still occurs, but the amount of useful information provided to the developer is maximized. Therefore, to provide this fine level of control, each module in UTL has a 32-bit debug variable, where each bit controls a particular subset of diagnostic output. A set of pre-processor macros (#defines) are provided so that the programmer can easily identify the subset of debugging output to which any given print statement belongs. At run time, if a particular piece of debugging output is *not* selected, the overhead for each

---

[41] `dbx` is part of Sun's *Workshop* development environment (www.sun.com)

[42] `gdb` is the GNU debugger provided by the Free Software Foundation (www.gnu.org)

debugging output block is only the time it takes to do a bitwise-and, a compare, and a branch operation. Furthermore, one can produce an optimized binary by simply substituting a “no-op” macro for each of the regular debugging macros. Having a means to remove the debugging output without deleting it from the source means that it will still be in the code in case it is needed again later—which, it invariably is.

3.7.4 Memory debugging macros

A frequent cause of problems in developing software in C or C++ is the manipulation of pointers. Network software in particular has this problem, because of the pointer arithmetic involved in efficient encapsulation and de-encapsulation of protocol data units (PDUs). Therefore, as part of UTL we developed routines to help us find two particularly nasty classes of bugs:

(1) memory leaks

(2) heap corruptions due freeing of dangling pointers

The basic approach is described in (Young, 1995.) We put a wrapper around the heap allocation and deallocation routines `malloc()` and `free()`. Our wrapper routines, `mem_coBlock`() and `mem_ciBlock()`, (memory check out block, and memory check in block) call the regular system `malloc()` and `free()` routines, but add a header to each piece of memory that is allocated. In the header of each chunk, we store a serial number, and the source code line number and filename of the statement that allocated the block of memory, and a “magic number”(a sentinel value

Table 3.7 Production releases of UTL (partial list)

Replace With Real Page 154!!!!!!!!!!!

used for data validation.)[43] We also keep track globally of how many pieces of memory have been allocated, and the total amount of memory allocated; these values are printed in diagnostic output controlled by the debug bits mentioned in the previous section. We then keep a binary search tree where the key is the pointer value itself; the tree contains a node for each chunk of memory that is outstanding.

Within UTL, any call to `malloc()` or `free()` is redirected via `#defines` to `mem_coBlock()` and `mem_ciBlock()`, respectively. The `mem_coBlock()` routine does a `malloc()` call for the requested amount of memory, plus the extra header, then adds the memory chunk to the search tree, and to the total of memory outstanding, and finally returns a pointer to the allocated chunk (past the extra header.) The `mem_ciBlock()` routine verifies that the pointer to be freed is in the tree; if not, it aborts the program with an error message, This message indicates that a `free()` was done on a dangling pointer (the standard run time library does not check for this, probably for reasons of efficiency.) On the other hand, if the block *is* found in the tree, then it is removed, the counters are decremented appropriately, and the real system free() routine is called.

Memory leaks are detected by turning on diagnostic output that is printed by `mem_ciBlock()` and `mem_coBlock()`. This output allows the developer to match up the `malloc`() and `free`() calls, and ensure that at the end of the program, the amount of outstanding memory is zero.

---

[43] The source code and line number come from the C pre-processor symbols `__LINE__` and `__FILE__`.

3.8 Related work

In this section, we survey related work in two areas: user-level (a.k.a. user-space) protocol implementation, and flexible protocol architectures.

3.8.1 User-level (user-space) protocol implementations

Previous work on user-level implementations of protocols frequently cites (Mogul et al., 1987) as a basis, which describes the Packet Filter, a facility in the kernel to provide efficient demultiplexing of incoming packets. The key idea of the Packet Filter is that demultiplexing is best done by a trusted process in the kernel, but that once the destination of the packet is determined, it should be delivered to a user-level process for the remainder of the processing. While this work is important for establishing the advantages of user-level implementation of upper layer protocol functions, because we rely on UDP for packet demultiplexing, the mechanisms in this work are not needed in our case.

(Thekkath et al., 1993) provides a summary of the arguments for building protocols at user level. It then describes a three component architecture for a user-space protocol implementation, and present performance results for a user-level TCP implementation built with their architecture. One component of their architecture is a protocol library linked in with the application, similar to our own UTL. The other two components are a registry server, which is single trusted user-level process per host, and a network I/O module that is located in the kernel.

This last component highlights a key difference between their work and ours: their architecture provides for user-level implementations of lower layer protocols such as IP and ARP in addition to TCP. As such, a more complex architecture involving some new elements in the kernel is required. Because we focus

only on transport layer processing, we are able to use the facilities of UDP and avoid having to extend the kernel in any way.

Other work in this area includes the following:

- (Edwards and Muir, 1995) which was already extensively discussed in Section 3.5.7.
- (Maeda and Bershad, 1993), which use essentially the same architecture as (Thekkath et al., 1993), but puts a greater emphasis on absolutely preserving the semantics of the original Berkeley Sockets API.
- The *x-Kernel*, which we discuss in the Section 3.8.3 on flexible protocol architectures.

3.8.3 Flexible protocol architectures

Several other researchers have studied environments for implementation of protocols. The most prominent is the *x-Kernel* developed at the Univ. of Arizona. (Hutchison and Peterson, 1988; Hutchinson and Peterson, 1991; O'Malley and Peterson, 1992.) The *x-Kernel* provides a framework for constructing protocols or protocol stacks from smaller microprotocols (similar to UTL layers). As in UTL, there is a standard layer-to-layer interface, and a meta-protocol defining the legal ways in which protocol can be composed. In these ways, UTL and the *x-Kernel* are similar.

However, the *x-Kernel* is a larger project in scope, and has different goals. UTL focuses exclusively on interactions between the application and transport layers, and provides a means to develop protocols at user level. The *x-Kernel* provides a means to build new in-kernel protocol stacks, along with an *x-Kernel* simulator allowing these stacks to be developed and tested at user level. It also provides an environment for building protocol stacks all the way from the data link layer, up through the network layer, to transport and application.

At the time we chose to develop our own library, the x-Kernel was less mature and less widely documented than it is today. Support for using the x-Kernel has increased dramatically, due to documentation and code available over the Web, and an introductory networks textbook based on the programming in the x-Kernel framework (Peterson and Davie, 1996). Therefore, as future work, we propose reexamining the architecture of UTL in light of the availability of the x-Kernel, and perhaps developing an x-Kernel based implementation of the functionality present in UTL. This would open up many new areas for investigation, since x-Kernel implementations of standard protocols (TCP, UDP) already exist.

Other work on flexible protocol schemes includes

- PascalCom which outlines an architecture based on a Pascal-like language (Tschudin, 1991)
- The ADAPTIVE project, a framework for experimenting with high-performance transport protocols, with an emphasis on multimedia applications and exploiting opportunities for parallelism in the protocol processing. (Schmidt and Suda, 1993; Schmidt et al., 1992)

3.9 Chapter summary, and future work related to UTL

This chapter described how the author designed and supervised the development of the Universal Transport Library (UTL), a tool for investigating flexible Transport QoS. UTL provides a framework for rapid prototyping of transport layer implementations, experimenting with application performance over a wide range of transport protocols.

We described the core principles that guided the UTL design and implementation: (1) avoidance of protocol-specific special-case code in the

application, (2) application level framing, (3) reasonable fallbacks, and (4) minimizing data copies.

We also described the means by which UTL provides a wide range of transport QoS to the application. The QoS provided is determined by the selection of a mechanism at connection establishment time, and by the setting of a set of parameter values. UTL mechanisms are composed of layers, and default parameter values. Layers provide the implementations of protocols and services. A set of formal rules is specified for composing mechanisms and determining the resulting QoS.

We also surveyed some of the challenges involved in the design and implementation of transport layer protocols in general, and user-level implementations in particular, including application-transport flow control, RTO calculation, and CPU scheduling.

Suggestions for future work on UTL already mentioned in this chapter include:

- Fully implementing the features of POCv2 not already in UTL, including the PR reliability class (Section 3.3.8)
- Finishing KX3 and evaluating it against other TCP-friendly protocols
- Completing the SRL layer (Section 3.3.8)
- Evaluating whether to migrate some UTL functions to the kernel, either directly, or through the x-Kernel (Section 3.5.1)
- Evaluating the penalty of message-oriented vs. byte-stream service (Section 3.5.4)
- Provision of a facility for QoS negotiation (Section 3.5.6)
- Redesigning the CPU scheduling to use POSIX threads (Section 3.5.7)

In addition, the following projects would be useful additions:

- Adding window size negotiation and mechanism negotiation to the KXx family of protocols. These features would aid considerably in automation of experiments, since currently, to test multiple window sizes and mechanism, it is necessary to have a separate server listening for each window size and mechanism under test.

- Adding the data preview feature described in Chapter 2 to the POL and TOL layers.

In the next chapter, we discuss the ReMDoR system, which has been the key application with which UTL has been tested.