A Configurable and Extensible Transport Protocol

Patrick G. Bridges[1], *Member, IEEE*, Gary T. Wong[2], Matti Hiltunen[3], *Member, IEEE Computer Society,* Richard D. Schlichting[3], *Fellow, IEEE,* and Matthew J. Barrick[1], *Member, IEEE,*

***Abstract*— The ability to configure transport protocols from collections of smaller software modules allows the characteristics of the protocol to be customized for a specific application or network technology. This paper describes a configurable transport protocol system called CTP in which microprotocols implementing individual attributes of transport can be combined into a composite protocol that realizes the desired overall functionality. In addition to describing the overall architecture of CTP and its microprotocols, this paper also presents experiments on both local area and wide area platforms that illustrate the flexibility of CTP and how its ability to more closely match application needs can result in better application performance. The prototype implementation of CTP has been built using the C version of the Cactus microprotocol composition framework running on Linux.**

***Index Terms*— Transport protocol, configuration, customization, extensibility.**

I. Introduction

Existing network transport protocols such as TCP [1] and UDP [2] have limitations when they are used in new application domains and for new network technologies. For example, multimedia applications sharing a network need congestion control but not necessarily ordered reliable delivery, a combination implemented by neither TCP nor UDP. Similarly, the congestion control mechanisms in TCP work well in wired networks but often over-react in wireless networks where packets can be lost due to factors other than congestion. The lack of appropriate guarantees or specific features has led to the widespread development of specialized protocols used in conjunction with or instead of standard transport protocols. These include IPSec [3] and SSL [4] for security, RSVP [5] for bandwidth reservation, RTP [6] for real-time audio and video, GTP [7] and CEP [8] for transport in Grid and high-end computing environments, and SCTP [9] for enhanced transport reliability. Developing such a protocol from scratch is, needless to say, often a significant undertaking.

[1]Computer Science Department, MSC01-1130, 1 University of New Mexico, Albuquerque, NM 87131, email: {bridges,barrick}@cs.unm.edu

[2]Computer Science Department, Boston University, 111 Cummington St., Boston, MA 02215, email: gtw@cs.bu.edu

[3]AT&T Labs - Research, Florham Park, NJ 07932, email:{hiltunen,rick}@research.att.com

In this paper, we describe our experience building a configurable transport protocol, CTP, that allows protocol semantics to be tuned to specific application needs without the engineering effort involved with new protocol development. With this approach, software modules that implement different service attributes or variants are written, and then a custom protocol is constructed by selecting appropriate modules based on the needs of the higher levels that use the service or on the specific characteristics of the underlying network or computing platform. Thus, for example, a congestion-control module can be configured together with a datagram service, or a security module can be configured together with other modules implementing a virtual circuit. The net result is, in effect, a family of transport protocols, each useful in a given scenario.

We experimentally demonstrate that CTP achieves comparable performance to existing protocols such as TCP and UDP on the applications for which they were designed. More importantly, we show that CTP can be customized for new applications to provide better performance than existing protocols without the software engineering overhead associated with developing a new protocol from scratch. Our prototype version of CTP is implemented using the Cactus microprotocol composition framework [10] running on UNIX UDP sockets on a cluster of Linux x86 machines and between x86 machines across the Internet.

The rest of this paper is organized as follows. In section II, we describe transport features that individual CTP modules would need to implement, describe the overall design of CTP in which configurable protocol modules (microprotocols) are implemented, and the relevant features of the Cactus protocol framework on which this design relies. Section III describes the microprotocols currently implemented in CTP and presents an in-depth example of how they interact to implement TCP-like protocol semantics in a particular CTP configuration. In section IV, we describe our experiences designing and implementing CTP, particularly how component decomposition and protocol infrastructure changed during implementation and testing, and our experiences using the Cactus to implement CTP. Section V illustrates the advantages of CTP's configurable approach to protocol construction by comparing its

performance in a variety of configurations to that provided to different applications by TCP and UDP. Finally, sections VI and VII compare CTP with related work, and present conclusions and directions for future work, respectively.

II. CTP Design

A. Transport Attributes and Algorithms

As a first step in developing a customizable transport protocol, we studied a wide range of transport protocols and identified various quality attributes that can be provided to higher levels and the algorithms used to implement these and other aspects of the service. In this case, we roughly divided quality attributes into the following:

- *Reliability.* Addresses the likelihood that the receiver receives all the data sent by the sender. Reliability can be increased by using different forms of redundancy ranging from retransmissions to the use of parallel channels to transmission of redundant data along one connection.
- *Ordering.* Describes guarantees concerning the ordering of data at the receiver relative to the order in which it is sent. For a stream-based transport services, the only reasonable ordering option is FIFO, but for message-based services other options may be reasonable.
- *Performance.* Describes how quickly data is transported from sender to receiver in terms of average throughput. The protocol may attempt to provide guaranteed performance by reserving resources or may do it only on a best-effort basis.
- *Timeliness.* Describes the timing characteristics of the end-to-end transmission with respect to maximum latency or jitter. Latency guarantees are typically made through resource allocation, while jitter can be controlled by adding a buffer at the receiver that is drained at a controlled rate.

TCP and UDP provide essentially a fixed set of these attributes. In particular, TCP provides strong reliability (guaranteed delivery) and ordering (in-order byte stream) semantics, but only best-effort performance and no timeliness guarantees. Similarly, UDP provides a best-effort performance, but with no ordering, timeliness, or reliability guarantees.

Given an attribute, numerous algorithms and protocols are often available for implementing its properties. For example, reliability can use some combination of positive, negative, or selective acknowledgment protocols, or several different forward error correction schemes. In some cases, different algorithms provide different types of guarantees. For example, IP-style one's complement and cyclic redundancy checks (CRC) provide integrity that protects against accidental data modification, while cryptographic methods such as DSA [11] protect against intentional modification. In other cases, different algorithms provide approximately the same guarantee, but with different trade-offs with respect to resource usage or other attributes. For example, forward error correction typically uses more bandwidth than acknowledgments, but usually provides faster recovery from failures, and thus a smoother data flow at the receiver.

Different choices can also be made for other design elements, such as whether to use congestion and/or flow control, and if so, what type. The protocol must also be able to interact appropriately with the protocol below it in the protocol stack. For example, messages may need to be fragmented into pieces or small messages coalesced into one packet. If a resource reservation protocol such as RSVP is available, the transport protocol may interact with it to make a resource reservation for the connection. Finally, the transport protocol must deal with such practical issues as connection establishment, monitoring, and tear-down.

B. Cactus

Cactus is a system for constructing highly-configurable protocols for networked and distributed systems. In this section we give a brief overview of Cactus; a detailed description of Cactus and its execution model can be found in [10].

Individual protocols in Cactus, termed *composite protocols*, are constructed from fine-grained software modules called microprotocols that interact using an event-driven execution paradigm. Each microprotocol is structured as a collection of event handlers and generally implements a distinct property or function of the protocol. Next, protocols are layered on top of each other to create a protocol stack using an interface similar to the standard *x*-kernel API [12] (e.g., `Demux()`, `Push()`, `Pop()`, `Open()`). This two-level approach has a high degree of flexibility, yet provides enough structure and control that it is easy to build collections of modules realizing a large number of diverse properties.

At runtime, composite protocol instances, termed *composite sessions*, are used to process packets. Composite sessions are created by protocol routines (e.g., `Demux()` or `Open()`) in response to open requests from either local applications or received packets. Each composite session contains a collection of *microprotocol instances* in which event handlers are bound to protocol-specific events to effect protocol processing.

Processing of structured messages by microprotocol-defined event handlers comprises the basic programming model of Cactus. Events are used to signify state changes of interest, such as "message arrival from the network". When such

an event occurs, all event handlers bound to that event are executed. Events can be raised explicitly by microprotocol instances or implicitly by the composite protocol runtime system.

The Cactus runtime system provides a variety of operations for managing events and event handlers. In addition to traditional blocking events, Cactus events can also be raised with a specified delay to implement time-driven execution, and can be raised asynchronously. Arguments can be passed to handlers in two ways, statically when the an event is bound to a handler and dynamically when an event is raised. Other operations are available for unbinding handlers, creating and deleting events, halting event execution, and canceling a delayed event. Handler execution is atomic with respect to concurrency, i.e., a handler is executed to completion before any other handler is started unless it voluntarily yields the CPU.

The Cactus message abstraction is designed to facilitate development of configurable services. One of the main features of Cactus messages are *message attributes*, which are a generalization of traditional message headers. Operations are provided for microprotocols to add, read, and delete message attributes. Furthermore, a customizable pack routine combines message attributes with the message body for network transmission (on-wire format), while an analogous unpack routine extracts attributes at the receiver.

Synchronization and coordination of execution activities in Cactus is accomplished through *event-based barriers* that may be associated with data items, including messages. A microprotocol instance can register with the barrier, and an event associated with the barrier will only be raised when all microprotocol instances registered with the barrier have entered the barrier. These barriers are used to coordinate activities across multiple microprotocols, especially to control the transfer of messages up and down the protocol stack.

C. Design Overview

In Cactus terms, CTP is a composite protocol in which each attribute or function described in section II-A is implemented by one microprotocol or a set of alternative microprotocols. Thus, the current design has one or more microprotocols for reliability, ordering, security, jitter control, congestion control, flow control, data and header compression, MTU discovery, message fragmentation and collation, and connection establishment, monitoring and tear-down. The goal of the design is to decouple the implementations of different attributes and functions to maximize the ability to mix and match different microprotocols to provide exactly the required properties. Decoupling the different features of transport protocols is not trivial, since often much of the functionality is tightly coupled for efficiency. For example, reliability, congestion control, and flow control in TCP often utilize the same transmission window data structure, while byte sequence numbers are used to implement reliability and ordering, and to provide necessary feedback for flow control.

Unfortunately, the design space for transport protocols is very large; to somewhat limit the scope of the problem, the current CTP design focuses on only bidirectional message-oriented point-to-point communication over an unreliable packet-oriented network protocol (e.g., IP). Specifically, an application uses a given CTP configuration to exchange arbitrary length messages (e.g., a video frame) with some application-defined semantics with a single endpoint. Since the design of CTP does not assume that the underlying network protocol supports such arbitrary length messages, microprotocols for fragmenting or coalesces messages into an appropriate transport unit—a *segment*—are provided. Finally, CTP addresses are currently local/remote IP/port number 4-tuples similar to those used by TCP.

D. CTP Events

As with all Cactus protocols, microprotocol instances in a CTP session interact using events that manipulate shared data—in particular, the messages and their attributes. CTP predefines a set of common events useable by all CTP microprotocols. These events are illustrated in figure 1. The figure uses solid arrows to indicate events raised by CTP's interface routines and dashed arrows to indicate causal relations between other events. For example, when the MSG FROM USER event is raised by CTP, some microprotocol will raise the SEGMENT FROM USER event. Additional local timeout events are used by several of the microprotocols. Most of the event names are self explanatory. For example, SEGMENT RECEIVED is raised when a previously sent segment is acknowledged, SEGMENT TIMEOUT when a segment's status has been unknown too long, and SEGMENT LOST when CTP is explicitly notified that a segment has been lost.

CTP makes extensive use of the Cactus event-based barrier, particularly event-based barriers associated with individual messages. For historical reasons, event-based barriers associated with messages are generally referred to as *hold bits*.[1] CTP uses three sets of hold bits on each message: send bits, deallocate bits, and done bits. The send bits are used to coordinate sending of segments down to lower layers and delivery of messages up to the application. A microprotocol

[1]The event-based barrier synchronization mechanism was created as a generalization of the hold bits originally associated with messages.

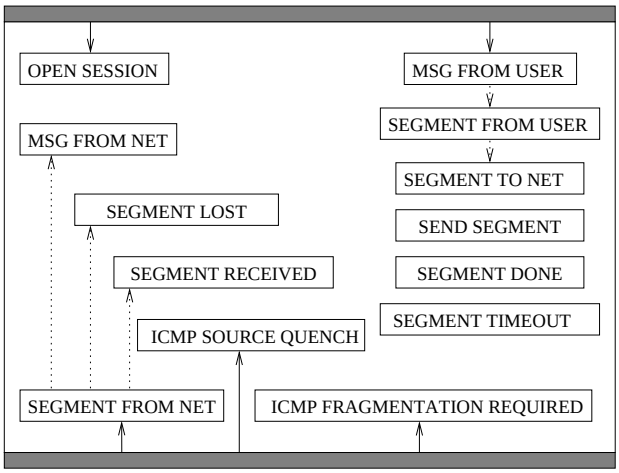


Fig. 1. Major CTP events.

sets a send bit in a given message when the message can be delivered up or down the graph as far as the microprotocol is concerned. When all of these bits are set, the message exits and the SEND SEGMENT event is raised so that microprotocols can be notified when a segment actually leaves the protocol. For example, congestion control, flow control, and reliability functionality in CTP each control send bits to determine when a segment can be transmitted, while flow control, jitter control, and the different ordering microprotocols use send bits to determine when a message can be delivered to the application.

Send bits allow different microprotocols to operate on messages independently without knowing which other microprotocols need to process the message. They also decouple the approval process from any kind of ordering—when all the required microprotocols have set their bits, the message exits the composite protocol independent of the order in which they were set. Note that systems supporting only hierarchical composition intrinsically dictate one fixed release order. Similarly, deallocate bits are used for determining when a segment will not be needed by any microprotocol in CTP and can thus be deleted.

Some microprotocols need to know when an outgoing message is not on the network (i.e., has either been acknowledged or timed out) *and* will never be retransmitted. For example, flow and congestion control microprotocols need to know when new capacity is available on the wire so that they can, for example, advance the trailing edge of the congestion control window. As this condition involves the agreement of multiple protocols, CTP uses per-message event barriers referred to as *done bits* to detect this condition. When every microprotocol participating in this barrier has entered the barrier, the SEGMENT DONE event is raised. Note that deallocate bits are not sufficient for this purpose because microprotocols may delay setting deallocate bits on messages even though they will never retransmit the packet. For example, the Forward Error Correction microprotocol delays deallocating sent messages so that it has the data needed to compute the contents of redundant packets using an erasure code algorithm.

E. Configuration and Initialization

Like most Cactus-implemented protocols, CTP composite sessions are created in response to an explicit open request from an application or when the `Demux()` protocol entry point receives a packet with a host/port 4-tuple that does not demultiplex to any existing session (addressing and demultiplexing in CTP is not currently configurable; this is an area of current work.) As with all Cactus protocols, the CTP session initialization routine is then invoked, resulting in the creation of session-global state, the instantiation of microprotocol instances, and the initialization of these microprotocol instances. At this time, microprotocol instance initialization routines set up their data structures and notify the runtime system of any necessary hold bits they will need on CTP messages. After the session and all of its microprotocol instances are initialized, the CTP demux routine raises the OPEN SESSION event in the new session so that microprotocols that perform connection establishment can execute appropriately. If the session was created as a reaction to a packet received from the network, the SEGMENT FROM NET event will be raised to allow processing of any data contained in the packet.

New CTP sessions select the appropriate microprotocol instances for each composite session based either on information in the locally-generated open request or on data in the packet that caused the creation of the new session. For local open requests, the current CTP implementation requires applications to specify exactly the microprotocols they desire in the session being created, including resolving dependencies by hand. Configuration tools such as those used in previous systems [13] could be used to ease this process, but that has not been a focus of our work thus far.

For open requests received from a remote host, CTP requires that packets that create a session contain sufficient data to determine which microprotocols were used to generate the received segment. In the most general case, connectionless protocols where any packet can establish a session, this is implemented as a 32-bit bitfield that is included with every CTP packet, with a different bit assigned to each possible CTP microprotocol. For connection-oriented CTP configurations, however, this bitfield need only be included in the connection establishment request.

Note that the microprotocol configuration in an existing

CTP session is not currently changeable at runtime. While feasible in principle, doing so would require substantial additional machinery either to quiesce the network or to support multiple microprotocol instances simultaneously while old packets are drained from the network. However, work to support such dynamic adaptation capabilities has been done in the context of Cactus [14] and in other systems such as the K42 operating system [15].

III. Microprotocol Highlights

This section gives an overview of some of the microprotocols available in the CTP suite, including those that implement reliable delivery, transmission control, message ordering, and jitter control. There are also several microprotocols that provide base functionality not directly connected with a specific semantic property. This section concludes with an example showing how individual microprotocols interact in a TCP-like configuration to implement TCP-like congestion control and reliability semantics on a sending node.

A. Base functionality

Transport Driver is the only microprotocol that must be present in any configuration. It adds port identifiers on all outgoing segments for demultiplexing and also contains trivial handlers for certain events to ensure that a message is carried through CTP irrespective of the presence of other microprotocols. It also sets the send bits to ensure that messages are sent even if there are no other microprotocols that set send bits in the configuration. The event interactions of Transport Driver are illustrated in figure 2. In the figure, arrows pointing to a microprotocol indicate that the microprotocol has a handler bound to the event and arrows originating in the microprotocol indicate that the microprotocol raises the event.

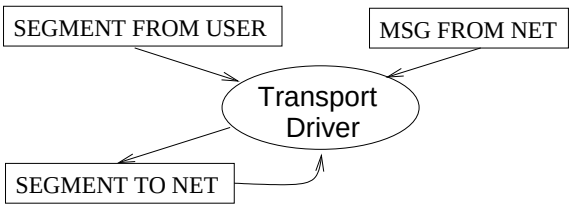


Fig. 2. Transport Driver event handling.

The Sequenced Messages and Sequenced Segments microprotocols add message attributes uniquely identifying each outgoing message and segment, respectively. While this labeling does not provide any service to the application, it is useful for other microprotocols such as reliability or ordering. Performing the procedure in a separate microprotocol allows the other microprotocols to share the same attribute, saving space in the message.

A group of microprotocols transforms messages into segments at the sender and then back to messages at the receiver. They are also responsible for raising the SEGMENT FROM USER and MSG FROM NET events. Fixed Size simply creates a separate segment from each message, Coalesce combines multiple small messages into one segment, and Resize fragments the messages into segments that can be handled by the underlying IP network without IP-level fragmentation (MTU discovery). One of these microprotocols must be present in each configuration.

Finally, a set of optional microprotocols is responsible for establishing and shutting down a connection, and for monitoring its status. Virtual Circuit implements a handshake protocol that provides reliable startup and shutdown semantics, and exchanges random initial sequence numbers for message and segment numbering. Virtual Circuit is completely transparent to other microprotocols, even those that use sequence numbers—if it is not included, constant initial values are used. The ability to realize this transparency stems directly from the Cactus event handling mechanisms. Specifically, this microprotocol binds event handlers to the SEGMENT FROM NET event, ordering them so that they are executed before other event handlers. When the event occurs, it uses a Cactus operation to stop the event, which prevents the other event handlers from executing. As a result, other microprotocols do not see any handshake messages and are unaware of the presence or absence of Virtual Circuit in a given configuration. An additional Keep Alive microprotocol is responsible for sending probe messages to detect link failures in the absence of application messages.

B. Informational microprotocols

CTP contains a number of microprotocols that collect information and provide it to other microprotocols by raising events or setting shared variables. These microprotocols can then use the provided information to make decisions. For example, the Round Trip Time Estimation microprotocol maintains an estimate of the end-to-end round trip time in a protocol-wide shared variable by handling the SEND SEGMENT and SEGMENT FROM NET events so that it can note when a segment is actually placed on the wire and when an acknowledgment for the segment is received. This estimate is then used by other microprotocols for detecting congestion and setting timeout values, for example.

The Positive ACK microprotocol is another, more complex, informational microprotocol used to track the status of transmitted segments. It implements a general cumulative acknowledgment facility necessarily more general than similar functionality in other protocols. In particular, it can be used in CTP configurations that do not require all messages to be delivered because it does not include any reliability functionality such as retransmission facilities. This allows it to be used, for example, in unreliable protocols that still need to track packet delivery status for flow and congestion control purposes as well as in reliable configurations that include microprotocols such as Retransmit.

This generality is achieved by slightly redefining the meaning of a cumulative acknowledgment and introducing a session-global data structure to decouple Positive ACK from the presence of reliability microprotocols. In Positive ACK, an acknowledgment indicates that the acknowledged segment was received *and* that the receiver no longer needs or expects to receive the acknowledged segment or any segment sent prior to it. Note that this does not mean that the previous segments were necessarily received—simply that they are unneeded, that is, that their *reliability constraints* have been met.

CTP's session-global data includes a data structure that keeps track of whether the reliability constraints on a each packet have been met. If a reliability microprotocol (e.g., Retransmit) is included in CTP, it sets the default reliability status of packets in this list to `RELIABILITY_UNMET` in its initialization routine, and then later sets it to `RELIABILITY_MET` when the packet is acknowledged. If a reliability microprotocol is not included in the configuration, however, the default reliability status of packets remains `RELIABILITY_MET`. This list allows Positive ACK and similar informational microprotocols to know for which packet to send a cumulative acknowledgment.

In reliable protocols, where the receiver expects to receive every packet, the more general definition of acknowledgments and the reliability tracking data structure results in the standard acknowledgment behavior used in protocols such as TCP. In protocols that do not require complete reliability, however, the more general definition of acknowledgments and the reliability tracking data structure allow acknowledgments for packets to be sent even if some previous packets have not been received. In addition, this design also allows for partially reliable configurations, where some packets must be transported reliably and some unreliably, although CTP does not currently include any microprotocols that make use of this flexibility.

Figure 3 shows how Positive ACK, Duplicate ACK, and an example reliability microprotocol (Retransmit, in this case) use

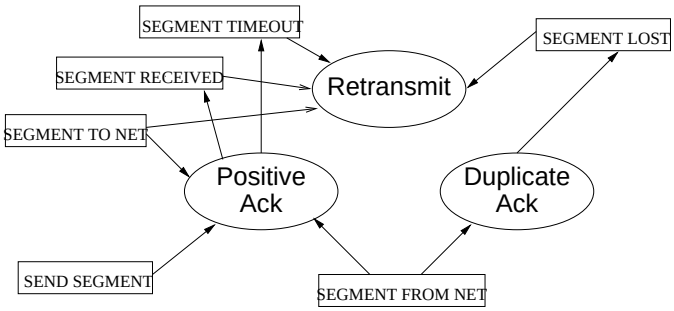


Fig. 3. ACK-related event handling.

events to track segment status. For each outgoing message (event SEGMENT TO NET), Positive ACK includes a cumulative acknowledgment attribute as described above, and also raises the SEGMENT TIMEOUT timer event when the message is actually transmitted (event SEND SEGMENT). For each incoming message (event SEGMENT FROM NET), it checks the acknowledgment attribute, and cancels the SEGMENT TIMEOUT event and raises the SEGMENT RECEIVED event if appropriate. Similarly, the Duplicate ACK microprotocol also monitors the SEGMENT FROM NET event and raises the SEGMENT LOST event when appropriate. Other microprotocols such as Retransmit then use these events and the data included in event arguments (e.g., sequence numbers) to determine when to retransmit old segments or release new packets to the network.

C. Reliability microprotocols

Reliable transmission can be implemented using different types of redundancy ranging from redundant network connections to redundant transmission over the same connection. CTP currently has two reliability microprotocols: Retransmit and Forward Error Correction. Retransmit is a traditional ARQ reliability scheme that relies on informational microprotocols to know when packets have been received and which ones should be retransmitted. As shown in the pseudocode in figure 4, it handles the SEGMENT LOST and SEGMENT TIMEOUT events and retransmits the appropriate segment when one of these events is raised. In addition, it allocates a done bit on each outgoing message and sets it upon receiving the SEGMENT RECEIVED event. As mentioned in section II, this allows other microprotocols to know when the message will not be retransmitted so that they can, for example, advance the congestion window.

Forward Error Correction transmits redundant data so that the receivers can reconstruct a complete transmission despite message losses. Forward Error Correction at the receiver then handles the redundant segments and uses them to create a new message for each of these missing segments and raises

```
micro-protocol Retransmit () {
  handler handleSegToNet(segment s){
    setSendBit (s);
  }

  handler handleRetransmit(int seq){
    m = HashLookup(protocolState.segmentHash, seq);
    clearSendBit (m); setSendBit (m);
  }
  handler handleSegRxd(int seq){
    m = HashLookup(protocolState.segmentHash, seq);
    SetReliabilityState (protocolState.relList, seq,
      RELIABILITY_MET);
    setDoneBit (m); setDeallocateBit (m);
  }
  initial {
    protocolState.defaultRelStatus = RELIABILITY_UNMET;
    requestDoneBit ();
    requestDeallocateBit ();
    requestSendBit ();
    bind(SEGMENT TO NET,handleSegToNet,0);
    bind(SEGMENT LOST,handleRetransmit,0);
    bind(SEGMENT TIMEOUT,handleRetransmit,0);
    bind(SEGMENT RECEIVED,handleSegRxd,0);
  }
}
```

Fig. 4. Retransmit microprotocol pseudo code

the SEGMENT FROM NET event for the reconstructed segments. Redundant data packets are also tagged with a special attribute to assure that they are not handed to the application. As a result, other microprotocols see the reconstructed segments as if they had arrived normally.

The specific error correction scheme currently used by this microprotocol is a block erasure code algorithm [16] that encodes k segments of original data into n segments of encoded data ($k < n$). At the sender, after k segments have been transmitted as normal, an additional $n-k$ redundant segments are computed and transmitted. The encoding scheme allows the receiver to compute all k original segments provided that at least k of the n segments are delivered intact. A number of tradeoffs are involved in selecting n and k in this system. For example, introducing a large percentage of redundant packets (i.e., $k/n \rightarrow 0$) makes the system more tolerant to losses, but lowers effective data bandwidth. Similarly, increasing n for a fixed value of k/n can increase resistance to burst losses, but also delays the delivery of redundant data and can place substantial computational load on the sender for constructing redundant packets.

Note that forward error correction and ARQ reliability can often be used together in the same CTP configuration. This gives the user a rich set of possibilities for reliable communication that can be used to match the specific requirements of particular applications.

D. Transmission control microprotocols

CTP offers flexible facilities for controlling the speed of transmission, typically used to ensure that a sender limits its outgoing traffic to a level acceptable to the network and receiver. Our architecture divides these microprotocols into two categories: flow control and congestion control.

a) Flow Control: Flow control refers to end-to-end transmission control that provides a mechanism for the receiver to dictate the sender's transmission speed. Available microprotocols include:

- *XON/XOFF.* The receiver issues suspend/resume instructions to the sender.
- *RTS/CTS.* The sender explicitly requests the ability to send more packets.
- *Windowed.* The receiver periodically informs the sender of its available buffer space.

These microprotocols all operate at the sender side by binding a handler to the SEGMENT TO NET event, which sets its send bit on an outgoing message only when restrictions on transmission are fulfilled.

At the receiver side, there are facilities in the API to allow higher level protocols to specify policies on traffic rates. The flow-control microprotocols can communicate this information to the sender either by transmitting new feedback messages to the sender or by piggybacking the information on existing messages. This feedback is handled at the sender in a handler bound to the SEGMENT FROM NET event.

b) Congestion Control: Congestion control behaves similarly to flow control in that it limits the transmission rate of senders, but is intended to avoid overrunning the capacity of the network rather than the receiver. Congestion control in CTP consists of two types of microprotocols: congestion control and congestion policy. Congestion control microprotocols are responsible for implementing the mechanism for controlling congestion, while congestion policy microprotocols describe corresponding policies. Typical configurations would include one congestion control and one congestion policy microprotocol.

Congestion control microprotocols, like flow control microprotocols, use send bits to provide a mechanism that regulates segment transmission. These microprotocols monitor protocol-wide shared variables that congestion policy microprotocols change in response to policy-specific indications of congestion. CTP currently implements two congestion control microprotocols: Windowed Congestion Control and Rate-based Congestion Control.

Windowed Congestion Control implements a simple window-based scheme that limits the number of unacknowl-

edged packets in the network. The size of the window is stored in a shared variable that can be changed by congestion policy microprotocols in response to various events. The Rate-based Congestion Control microprotocol works similarly, but instead controls the average outgoing byte rate based on a shared variable. Because each congestion control microprotocol uses a different send bit for controlling segment transmission, multiple congestion control microprotocols can be used simultaneously when appropriate.

As already mentioned, congestion policy microprotocols in the current design work by changing shared variables exported by congestion control microprotocols. As such, these microprotocols are designed to work with specific congestion control microprotocols. Available congestion policy microprotocols include:

- TCP Congestion Detection. This microprotocol handles the SEGMENT RECEIVED, SEGMENT TIMEOUT, and SEGMENT LOST events and changes the congestion window used by the Windowed Congestion Control in response to these events in accordance with the congestion control policy used by TCP [17]. Note that the policy implemented by this microprotocol does not depend on the presence of the Retransmit microprotocol in the CTP configuration, so it may be used with unreliable communication or in combination with Forward Error Correction.
- TCP-Friendly Rate Control. This microprotocol monitors segments status events and sets the maximum outgoing data rate used by Rate-based Congestion Control according to the TCP response equation [18].
- SCP Congestion Detection. This microprotocol monitors the average round-trip time and the packet status events and sets both the outgoing data rate used by Rate-based Congestion Control and the window size used by Windowed Congestion Control similarly to the SCP protocol [19].

Other congestion policy microprotocols, for example, ones that use ICMP source quench messages or ECN notification bits [20], are also easily implemented in this framework.

E. Ordering and jitter control microprotocols

Ordering microprotocols are relatively simple for point-to-point communication as currently supported by CTP. The sender can add a message attribute that indicates the order of the message either as a sequence number or by specifying the message's logical predecessor(s). The current implementation has a Reliable FIFO microprotocol, which enforces strict in-order delivery by buffering out-of-order messages and sending them to the application only after their predecessors have been delivered, and a Lossy FIFO alternative that discards messages that arrive out of order after a configurable delay. A Semantic Order microprotocol uses ordering information provided by the application to record and enforce the logical predecessors of each message. An Out of Band microprotocol can be used with any ordering microprotocol to allow urgent out-of-band messages to be delivered as quickly as possible by overriding the send bit used by the current ordering microprotocol.

Jitter control microprotocols are structurally similar to ordering microprotocols, but use the passage of time rather than predecessor information to decide when the send bit in a message is set. These microprotocols include Fixed Rate Jitter, which delivers messages separated by a fixed time interval and Timestamp Jitter, which preserves the sender's time intervals between messages at the receiver.

IV. Design and Implementation Experiences

Over the course of designing and implementing CTP, we gained substantial experience in dealing with configurability in CTP, as well as using the Cactus protocol framework. As part of this process, we ran into issues with our original CTP design that we had to resolve. In this section, we discuss our experiences designing and implementing a configurable protocol, the mistakes made in this process and how they were remedied, and our experiences using the Cactus event-based protocol framework for implementing a substantial configurable protocol.

A. Configurability and extensibility in CTP

To make CTP highly configurable, the different microprotocols have been designed to be as independent as possible. However, there are some *dependencies*—when one microprotocol requires that another be in the configuration to function correctly—and some *conflicts*—when two microprotocols cannot be in the same configuration. The dependencies in the current design are relatively simple. Every configuration must have Transport Driver and one of the message-to-segment conversion microprotocols Fixed Size, Coalesce, or Resize. The reliability and FIFO ordering microprotocols use sequence numbers provided by the Sequenced Messages and Sequenced Segments. Similarly, most flow and congestion control microprotocols require an informational microprotocol such as Positive ACK to provide feedback on the status of transmitted segments. Finally, congestion control policy and mechanism microprotocols must be used in conjunction with each other.

Conflicts are either syntactic or semantic in nature. An example of a syntactic conflict is that only one message-to-

segment conversion microprotocol should be in each configuration, while an example of a semantic conflict is that Lossy FIFO and a reliable communication microprotocol should not be used together. Semantic conflicts do not cause the combination to fail, but the resulting semantics do not satisfy the properties of both of the microprotocols.

Despite these dependencies and conflicts, there are still hundreds of possible different CTP configurations even with a small number of different microprotocols for each transport property and function. The challenge is to identify the correct configuration for each application domain and execution environment. In many cases, this may require experimentation with different combinations to reach the optimal one.

CTP is also designed to be easily extensible, meaning that new microprotocols can be added without modifying the existing ones. The actual effort needed depends on the type of extension. It is typically trivial to add a new alternative implementation for an existing property or function, since the event and data structure interactions are usually the same as in existing microprotocols.

On the other hand, adding a completely new property or function can be more difficult. The implementor must first determine if CTP already has all the necessary events required by the new microprotocol. If not, the CTP framework or some of the existing microprotocols may need to be modified to raise these events. However, completely new microprotocols can often be implemented using the existing set of events. For example, in our design, the jitter control microprotocols were added after the rest of CTP was designed with no modifications to other microprotocols.

B. Corrected Design Mistakes

Over the course of designing and implementing CTP, we ran into two substantial design mistakes that required re-architecting parts of the system. In particular, complex protocol services implemented monolithically in protocols such as TCP initially led us to make similar monolithic services in CTP that either did not sufficiently decompose complex services, or did not separate mechanism and policy decisions. This resulted in an insufficiently flexible protocol when we initially tried to use CTP for multimedia applications like the one used in the experiments in section V. These two issues are discussed in more detail below.

Decomposing Complex Interactions. Flexible configurability in CTP did not come without substantial effort. For example, while the reliability, ordering, and flow control transport functions that are tightly connected in TCP are completely independent in our final design, this was not originally the case. In our original design, a single Positive ACK microprotocol performed two logically separate functions: tracking the status (*received*/*lost*/*timed out*) of transmitted segments, and reliable transmission of segments using timeouts and retransmissions. This overloading, the result of failing to completely decompose acknowledgment functionality inspired by TCP, caused problems for applications that wanted segment status tracking but not retransmissions such as streaming multimedia transmission applications.

Decoupling these responsibilities required the introduction of several new microprotocols and events. Much of this decoupling comes from the use of Cactus' event-based programming model, but some required the generalization of protocol functionality and the introduction of additional mechanisms and data structures. We decomposed the original Positive ACK microprotocol into several microprotocols, namely Positive ACK, Duplicate ACK, Negative ACK, Retransmit, and RoundTripTimeEstimation. We also introduced three new events, SEGMENT RECEIVED, SEGMENT LOST, and SEGMENT TIMEOUT, to announce when segments are acknowledged, explicitly lost, or have had an unknown status for an unacceptable amount of time. This decomposition allowed CTP to be configured to use acknowledgments for feedback about segment arrival and loss without mandating the introduction of retransmissions and their negative effects on multimedia applications.

The new Positive ACK microprotocol implements acknowledgments and segment timeouts, while the Negative ACK and Duplicate ACK add additional packet tracking functionality. On the sender side, all of these microprotocols work by raising the appropriate events at the appropriate time; these events are then responded to by Retransmit. On the receiver side, Positive ACK was changed to acknowledge packets when it has either received a packet or no longer needs a packet, and a session-global data structure describes whether the reliability constraints on each received packet have been met. In reliable protocols, where the receiver expects to receive every packet, this behavior results in the standard acknowledgment behavior used in protocols such as TCP. In protocols that do not require complete reliability, however, the more general definition allows an acknowledgment for a packet to be sent even if some previous packets have not been received. Note that this change also required the introduction of done bits for use by WindowedCongestionControl as described in section III-B.

Separating Mechanism and Policy. Another shortcoming of the original CTP design was that it did not separate congestion control mechanism and policy. As in all systems, keeping such separation is important, and failing to do so in our original CTP design was a substantial mistake. This problem

was solved by introducing two different microprotocols that implement congestion control mechanisms, WindowedCongestionControl and RateBasedCongestionControl, and a variety of different microprotocols that implement different congestion control policies as previously described. The most substantial change required by this generalization was the introduction of the done bits on each CTP segment and the corresponding SEGMENT DONE CTP event, allowing WindowedCongestionControl to advance the trailing edge of the congestion window at the appropriate time.[2]

As a result of this experience, separate policy microprotocols were similarly used for controlling forward error correction parameters when CTP was later modified to support adaptation of error correction parameters. To further enable careful separation of mechanism and policy later work on a system named Cholla [21] explicitly separated protocol policies into a separate policy control engine where they could be separately composed, controlled, and analyzed.

C. Cactus Event Experiences

After implementing a variety of CTP microprotocols and testing a variety of different configurations, we found the largest source of bugs was in the ordering of event handlers. Cactus allows event handlers to bind with different *order priorities*, and handlers are run in numeric order priority. Excessive use of event ordering, however, resulted in a number of different bugs. In the original implementation for example, there were not separate SEGMENT TO NET and SEGMENT SENT events; microprotocols that wanted to run after segments were sent would simply bind to SEGMENT TO NET with a large order priority. As new microprotocols were introduced, however, misorderings between when handlers were run could cause, for example, round trip times to be calculated inappropriately.

To address this problem, later implementations of CTP were changed to use more fine-grained events instead of ordering among event handers on fewer events. The resulting definition of more CTP events along the sending and receiving processing path required us to understand and interface with longer event chains when implementing new protocols. However, our experience shows that documenting and understanding the (well-defined) longer event chains was much easier than understanding somewhat shorter event chains and the ordering constraints of every possible microprotocol in the system.

[2]Note that simply monitoring the SEGMENT RECEIVED, SEGMENT TIMEOUT, and SEGMENT LOST events is not sufficient for this purpose. This follows because a packet may or may not be reintroduced into the network depending on whether or not, for example, the Retransmit microprotocol is included in the current CTP configuration.

V. Experimental Results

A. Overview

While CTP cannot compete at this stage with tuned versions of TCP and UDP, the flexibility provided by the service is useful for application domains and execution environments that are not the focus of the standard protocols. In particular, CTP is useful when either a set of characteristics that falls somewhere between TCP and UDP is required, or for cases where stronger guarantees are needed than TCP provides. CTP is also appropriate when there is the opportunity to configure a protocol to match the characteristics of a specific network environment. The goal of this section is to quantify the potential overheads and benefits provided by the configurability of CTP.

In the remainder of this section, we present local area and wide area network results in a variety of situations. Local area performance results were collected between two 2-processor 2.2 GHz Pentium 3 Xeon machines running Linux kernel 2.4.18 across a quiescent 100 Mbps Ethernet; only one processor was used by the test program. The C implementation of Cactus 2.2 was used for composing microprotocols into a composite CTP protocol running at user level on top of Linux UDP sockets. Note that this imposes additional overhead on CTP compared to TCP and UDP. Wide area performance results were collected between Linux machines at the University of New Mexico (UNM) and the Georgia Institute of Technology (Georgia Tech).

Section V-B uses these platforms to quantify the cost of configurability in CTP by comparing latency and bandwidth numbers in different CTP configurations over both local and wide area networks. Section V-C then illustrates the potential benefits of CTP by customizing protocol configurations to application-specific and hardware-specific needs.

B. Configurability Overhead

The first set of experiments measures the bandwidth and ping-pong latency of UDP, TCP, and various configurations of CTP. Four different CTP configurations are included:

- CTP-Minimal: a minimal CTP configuration containing only the driver and fragmentation/reassembly microprotocols.
- CTP-LossyFIFO: the minimal CTP configuration augmented with per-message sequence numbers and unreliable in-order message delivery microprotocols.
- CTP-Video: a CTP configuration for video transmission that uses SCP-style congestion control, positive and negative acknowledgments, round-trip-time estimation, and in-order unreliable message delivery.

- CTP-Bulk: a TCP-Tahoe-like CTP configuration including reliable, in-order message delivery using retransmissions, duplicate acknowledgments, and TCP-style congestion control.

Note that the first three of these configurations are all unreliable configurations; only CTP-Bulk guarantees reliable transmission of all data.

In the latency tests, two machines ping-pong minimal-sized application packets 10 times to measure the average round-trip latency for one round trip. In the bandwidth tests, a sending application transmits 1000 1250-byte messages to a receiver, which replies with a user-level acknowledgment once all the data has been received. We measure the interval at the sender between the transmission of the first packet and receipt of the acknowledgment and use this to compute the end-to-end data transmission rate. To enable direct comparison of protocol processing costs, the PUSH flag is set on every message handed to TCP, causing it to preserve message boundaries and send the same number of data segments as the other protocols; we confirmed experimentally that the same message boundaries were used in TCP.

Table I shows the averages and standard deviations of 10 runs of the bandwidth and latency tests on both local and wide area networks, with the top part of the table comparing unreliable protocol configurations and the bottom part comparing reliable protocols. All measurements were made on the receiver after several initial packet exchanges to allow the congestion control window to open fully.

These results indicate a latency overhead of approximately 100 microseconds per round trip over UDP in the simple local-area test and execution environment, with approximately the same service guarantees. Similarly, bandwidth is competitive with UDP, although slightly less because this version of CTP is layered on top of UDP and because of protocol overhead such as the longer CTP headers required to support the sophisticated semantics of more complex configurations (68 byte CTP headers as opposed to 8 byte UDP headers). CTP header overhead is currently unoptimized, however, and can be reduced by specializing headers to particular configurations instead of having a single generic header that encompasses all current possible CTP configurations. Additionally, running CTP directly on top of IP would lower its latency costs significantly.

As microprotocols implementing more complex semantics are added to CTP configurations in the first part of the table, latency gradually increases and bandwidth slightly decreases. Adding relatively simple microprotocols such as Lossy FIFO and Sequenced Messages to the CTP configurations (the CTP-LossyFIFO configuration) adds negligible overhead; more complex microprotocols that implement, for example, congestion control, introduce correspondingly more overhead.

In the wide area unreliable results, latencies are dominated by wide area network costs, which obscure event overhead costs. Bandwidth numbers vary as expected, with the UDP, CTP-Minimal, and CTP-LossyFIFO configurations providing the best bandwidths given their lack of congestion control. CTP-Video provides less bandwidth because of congestion control actions, but more bandwidth than the TCP and CTP-Bulk configurations. Again, this is expected since the SCP-based congestion control policy used by the multimedia-oriented CTP configurations is more aggressive than TCP-derived policies and known to not be TCP-fair.

Comparing the reliable protocols, the latency overhead of CTP-Bulk compared to TCP is somewhat higher, on the order of 200 microseconds. This is caused by the increased event processing in CTP for the complex configuration required for full reliability. We expect the latency performance of all CTP configurations to improve as the event mechanisms in the Cactus runtime are optimized, although it is probably unrealistic to expect CTP to beat TCP and UDP for this type of use.

The bandwidth differences between CTP-Bulk and TCP are caused by minor differences in delayed acknowledgment handling and packetization in the two protocol implementations. Specifically:

- CTP-Bulk currently has an MTU of 1250 bytes as opposed to the 1400+ bytes that TCP uses, has larger headers, and runs on top of UDP.
- TCP (as a stream protocol) maintains the sender window sizes in bytes, while CTP-bulk maintains a window size in packets, since it is a message-oriented protocol.

These two differences prevent CTP-Bulk from utilizing the bandwidth of a lower bandwidth wide area connection as effectively as TCP does. Note, however, that CTP's modular structure makes such differences easy to change when appropriate.

C. Benefits of Custom Configurations

CTP can be tuned to provide optimized behavior for given applications or hardware environments similar to hand-built custom protocols without the engineering overhead of developing such protocols from scratch. In this section, we demonstrate the performance benefits that customizing CTP configurations to application- and hardware-specific needs can provide.

Protocol	Local Area		Wide Area	
	Latency (ms)	Bandwidth (Mb/sec)	Latency (ms)	Bandwidth (Mb/sec)
UDP	0.152±0.0028	90.28±0.005	413±3	61.54±1.317
CTP-Minimal	0.273±0.0052	81.76±0.014	414±0.4	56.81±0.637
CTP-LossyFIFO	0.279±0.0031	81.77±0.008	414±0.7	56.62±0.975
CTP-Video	0.366±0.0050	82.17±0.005	414±1	21.97±0.745
TCP	0.162±0.0029	89.40±0.025	412±0.9	9.38±0.189
CTP-Bulk	0.380±0.0090	68.31±0.148	415±0.8	4.73±0.288

TABLE I

LATENCY AND BANDWIDTH COMPARISON

1) Application-Specific Customization: To study the potential application-level benefits of protocol customization, we ran CTP as the underlying transport protocol for a custom Cactus multimedia-transmission and playback application. This application sends compressed audio or video to a remote receiver, which then plays back the received data in realtime from a playback buffer with fixed time capacity. This application supports both uncompressed and compressed (H.263/Ogg Vorbis) audio and video streams.

We studied the impact that custom CTP configurations have on an audio transmission configuration of this application using UDP, CTP-Bulk, and a new configuration CTP-Audio for audio transmission that is configured identically to CTP-Video except for the addition of a block-erasure forward error correction microprotocol. CTP-Bulk acts as a proxy for TCP performance in this experiment, since we did not have the kernel-level access that would be needed to vary the loss experienced by the TCP protocol on the wide-area test machines. Audio packets were sent at 128kbps on both low-latency (local) and high-latency (wide-area) networks, and with different amounts of additional packet loss at the ingress network device to examine how different protocol configurations and network conditions affected application performance. The application was set to use a fixed 3000ms playout buffer, and CTP-Audio was set to use N=5 and K=4 to be able to recover from one dropped data packet out of every five packets. Each test consisted of 1500 packet transmissions, and was conducted 10 times on each protocol/network configuration.

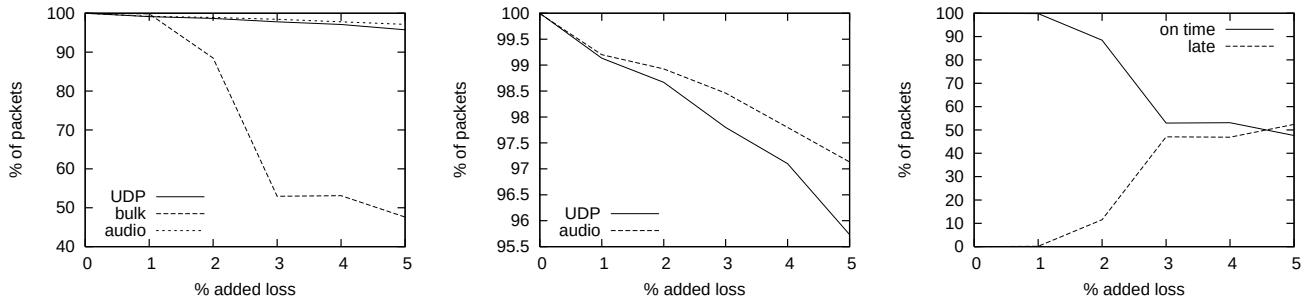
Figure 5(a) shows the performance of all three protocols on this application in terms of the percentage of packets delivered within the application playout window on a wide-area network between UNM and Georgia Tech. CTP-Bulk is unable to deliver packets on time in the face of significant packet loss, while UDP and CTP-Audio continue to provide reasonable service to the application. Figure 5(b) shows only UDP and CTP-Audio performance over wide-area networks, and demonstrates that CTP-Audio is able to deliver packets on time more robustly that UDP in the face of packet loss. Local-area comparisons between UDP and CTP-Audio behave essentially the same.

Figure 5(c) provides a more detailed breakdown of the performance of the CTP-Bulk protocol in the wide-area case. Since CTP-Bulk delivers all packets in order, as packet loss increases, packets are delivered increasingly late due to the TCP-like retransmission-based reliability scheme. UDP and CTP-Audio, on the other hand, deliver packets in a timely fashion. All of the late packets shown in 5(b) and 5(c) are due to packet loss, though CTP-Audio delivers more packets on time in the face of packet loss thanks to the forward-error correction service it provides to the application.

Of course, existing protocols, for example RTP [6] and SCTP [9], can provide application benefits similar to those shown above. However, each of these protocols had to be constructed from scratch, and are not easy to modify to support other, different application needs. CTP, however, allows the application authors to customize protocol behavior using a single integrated package that already supports a wide range of application-desirable semantics.

2) Hardware-Specific Customization: In the previous case, CTP was able to be easily reconfigured to provide superior performance to applications compared to TCP and UDP because the service requirements of the application were different than those provided by TCP and UDP. However, CTP configurations can provide superior performance compared to TCP even in cases where TCP exactly matches application service requirements, particularly when the underlying network hardware violates fundamental assumptions that TCP makes.

For example, modern versions of TCP derived from the BSD code retransmit segments after receiving 3 duplicate ACKs or upon expiration of a retransmission timeout, However, the TCP retransmission timer is typically very coarse, on the order of 500ms. Local wireless networks, connections across campus networks or even wide area-networks frequently yield round-trip times on the order of tens of milliseconds or



(a) UDP/CTP-Bulk/CTP-Audio Wide-Area **(b) UDP/CTP-Audio Wide-Area** **(c) CTP-Bulk Breakdown**

Fig. 5. Real-time Streaming Media Performance of UDP, CTP-Bulk, and CTP-Audio

Loss rate (%)	Round-Trip Latency (ms)	
	CTP-Bulk	TCP
0	0.34±0.002	0.17±0.0008
1	0.62±0.07	2.1±8
2	0.66±0.07	4.4±8
3	1.9±0.5	8.0±13
4	2.1±0.4	13±18
5	2.4±0.7	16±25

TABLE II

ROUND-TRIP LATENCY ON LOW-LATENCY UNRELIABLE NETWORKS

faster, so faster retransmission timers can be beneficial under certain circumstances. This is particularly true on, for example, 802.11b wireless networks, which can have low latencies and high drop rates.

We have measured the performance of CTP using the CTP-Bulk configuration described above. This configuration includes the Positive ACK and Retransmit microprotocols described in sections III-B and III-C, which use fine-grained retransmission timing. Table II lists the average round-trip latency of this CTP configuration compared to TCP. These times were measured using 10 tests of 100 back-to-back round-trips using zero-length application packets. This test was performed on the same platform as described above; network packet losses were simulated by randomly dropping varying proportions of packets on each receiving machine.

Although TCP has better latency in the lossless case, CTP was able to provide faster delivery on average when losses occurred by retransmitting more quickly. CTP can provide similar advantages in other environments where TCP is known to perform sub-optimally, such as high bandwidth-delay product links and long-distance wireless networks, or networks where losses may not be the result of congestion but may instead indicate, for example, radio interference. Moreover, CTP allows the user to configure all of these from a single integrated package, rather than forcing the construction of new specialized protocols from scratch.

D. Performance optimizations

The performance of a composite protocol built using Cactus such as CTP can be optimized in any number of ways. These optimizations can be classified based on whether they require changes in the Cactus runtime system or microprotocols, and the extent of these changes. The least intrusive optimizations customize the protocol's behavior using features in the Cactus runtime specifically provided for such customization. For example, message handling operations can be customized to construct message headers in whatever format is most efficient for the particular protocol by customizing the message pack and unpack routines as mentioned in section II-B.

Another type of optimization modifies the Cactus runtime system, but does not require changes in the microprotocol code. For example, to eliminate the table lookups required to invoke customizable operations, the message handling operations can be added as static functions to the runtime system. Similarly, event dispatch and handling performance could be dramatically improved using techniques already demonstrated elsewhere [22], [23].

Finally, some optimizations require that the chosen microprotocols be modified in some way, either by hand or through automatic compile time or run time optimization. For example, the indirection required to raise an event can be optimized by replacing the raise operation with direct calls to the appropriate event handlers or even by inlining the handlers.

In the experiments above, the only optimization used was the first one described above, where the Cactus message handling operations are customized. This optimization resulted in a minor decrease in the latency and increase in the bandwidth in the CTP-Minimal and CTP-Bulk configurations over unoptimized CTP. This only took a few hours of programming and did not require any changes to the Cactus framework

or CTP microprotocols. Other work has shown that more aggressive CTP optimizations can substantially improve CTP bandwidth performance [24].

E. Limitations

The current version of CTP has a number of limitations, mostly because it was designed primarily to serve as a prototyping environment for testing different combinations of transport-related algorithms and functions. One is that it uses a push/pop interface for interacting with upper levels rather than a more standard socket-type API. It is, of course, possible to add such an API on top of CTP. However, the traditional socket interface must be extended somewhat to support specification of the desired transport properties for each connection and possibly for each message (e.g., semantic ordering, out of band).

A second limitation is that CTP is not interoperable with any standard transport protocol, partially due to its use of the custom message format. However, limited interoperability could be added easily by customizing the message packing routine to generate a message header format compatible with an existing protocol such as UDP. Note, however, that enforcing compatibility with other protocols such as TCP would place severe restrictions on CTP's configurability, since its semantics would be limited to those provided by the existing protocol.

VI. Related work

A. Composite Transport Protocol Systems

Several other researchers have explored composite protocol frameworks, generally in the context of specialized environments. Specifically, XTP [25] and TP++ [26] have been used to support flexible data transport in high-speed networks, and Minden's composite protocol system [27] supports transport protocol composition for active network systems. XTP, for example, can be configured to support different amounts of reliability and different connection establishment mechanisms, while TP++ explicitly supports the semantic needs of three different classes of applications: latency-sensitive applications, bulk data transport applications, and distributed transaction systems.

In contrast, CTP is designed to allow very general configurability, enabling its use in a wide range of general purpose and specialized applications, as well as for prototyping new protocols and networking instruction.[3] Unlike these systems, CTP also allows the wire message format to be customized, potentially enabling backwards compatibility with protocols such as TCP and UDP. When implemented, this backwards compatibility could enable the broader adaptation of CTP in production, research, and teaching environments, something that existing fine-grained protocol composition systems have yet to achieve.

[3]CTP was used for student protocol implementation in a Computer Networking class at the University of Arizona in 2003, for example, and we are currently exploring using it for these purposes more broadly.

B. Protocol Composition Frameworks

A number of different configuration frameworks have been used to construct modular, and to some degree configurable, transport services. Most of these frameworks use a linear or hierarchical composition model where the communication subsystem is constructed as a stack or directed graph of modules with identical interfaces for interchangeability. Module interactions in these systems are typically limited to message exchange between adjacent modules in the composition graph. Examples of such composition models include the System V STREAMS [28], the *x*-kernel [12], CORDS [29], Horus [30], Ensemble [31], Globus XIO [32], and Rwanda [33]. To our knowledge, none of these approaches have been used to construct transport services that are customizable to the same degree as CTP.

The Cactus model provides a more flexible framework for constructing configurable transport protocols than any of the hierarchical models. Cactus does not force a linear order between modules when the modules are logically on the same level or even completely independent. For example, reliability, flow control, and congestion control in our design are independent, which means that no specific execution order is needed or enforced by Cactus. Furthermore, Cactus allows arbitrarily rich interactions between modules that need to interact rather than limiting interaction to be message exchanges between adjacent modules in a graph. Finally, modules in hierarchical models typically must act as data filters that get one chance to process a message when it traverses the communication subsystem. In Cactus, a microprotocol can be much more sophisticated and can process a message at multiple places as it traverse the composite protocol. For example, a microprotocol may be notified when a message arrives at the composite protocol, when it is ready to be transmitted, and when it has been sent out. As a result, microprotocols do not need to be simple data filters, but can implement arbitrary logical transport properties.

Protocol heaps [34] propose a non-hierarchical *role-based* approach to constructing network services and suggest that such a role-based approach could be used either in a single layer of the protocol stack or to replace protocol stacks altogether. The authors note that roles in their proposed system

are similar to microprotocols in Cactus' predecessor Coyote [35], although Cactus and Coyote focus on non-hierarchical composition inside a layer of the network stack as opposed to across an entire protocol stack. To our knowledge, protocol heaps have not yet been used to implement substantial protocols such as we have done with CTP. However, because of the similarities between the two systems, our experiences designing and implementing a flexible, non-hierarchical transport service and our measurements of the costs of such flexibility should be directly applicable to protocol heaps.

Adaptive [36] introduces a non-hierarchical approach for constructing configurable protocols. In this approach, each protocol or service consists of a "backplane" with slots for different protocol functions such as flow control and reliability. The fact that a service is pre-divided into a fixed set of functions (or slots) naturally restricts the composition, e.g., slots cannot be left empty and new slots are difficult to add. Interactions between different protocol functions is also prescribed by the backplane. Adaptive provides a higher level configuration interface, where a protocol composition is created automatically based on a functional specification. Such a higher-level interface could also be developed for CTP if desired.

STP [37] focuses on operating system and network safety in deploying new network protocols. STP protocols are written in Cyclone, a type-safe version of C, and distributed in source form over the Internet. The operating system then uses TCP-friendly rate-control to guarantee the network safety of the resulting protocol. This combination of sandboxed code and resource control eases the deployment of novel networking services. Similar techniques could be used to deploy CTP microprotocols to guarantee the safety of the resulting composite protocol.

C. Other Novel Transport Protocols

A number of other transport protocols have been proposed since the introduction of TCP and UDP. Examples include the Reliable Data Protocol (RDP) [38], [39] that provides a message-based transport service with reliability and optional FIFO ordering guarantees, and the Versatile Message Transaction Protocol (VMTP) [40] for transactional (RPC style) communication. The latter has certain customizable features, including optional security and customizable reliability and some support for real time and multicast data. More recent proposals include the Real-Time Transport Protocol (RTP) [6] that supports transmission of real-time data such as audio or video over multicast network services, the Stream Control Transmission Protocol (SCTP) [9] that provides improved reliability using techniques such as multihoming, and partially-reliable transport protocols for use in multimedia services [41]. Extensions to TCP have been developed to improve its performance and applicability for specific application or execution domains. Examples of such extensions include selective acknowledgments [42] and support for transaction-oriented services [43].

The goal of CTP is not to be yet another transport protocol or yet another TCP extension. Rather, CTP is a prototype of a completely customizable transport protocol that can be configured to serve any application domain in any execution environment to the best degree possible. CTP can also be used as a prototyping environment for testing new algorithms for different transport properties in different execution environments.

VII. Conclusions

The ability to customize transport protocols can provide important flexibility when it comes to supporting new applications and new network technologies. In this paper, we have described an approach to building such services based on the Cactus design and implementation framework, as well as a concrete realization of the approach in the form of CTP. In this family of transport protocols, various attributes such as reliable transmission and congestion control are implemented as separate microprotocols, which are then combined in different ways to provide customized semantics. Initial experimental results indicate that, while the performance is somewhat slower than TCP and UDP for similar configurations, the ability to target the guarantees more precisely can in fact result in better performance. While it will always be possible to construct a specialized solution that performs at least as well as CTP, CTP allows easy component-based construction of custom transport protocols with minimal effort.

Other future work will concentrate in three areas. First, we intend to use CTP as an experimentation and prototyping platform to implement and measure different transport-related algorithms. This work will require extensions to CTP on configurable (non-IP-centric) addressing and demultiplexing. Support for MPI-style matching instead of port-based demultiplexing is just one example of this. Support for other addressing and demultiplexing schemes, for example, virtual channel identifiers (VIDs) are also being considered. Second, we plan to extend CTP to support customizable multicast and group communication. Finally, we will explore further performance optimizations, both in the CTP composite protocol and in the Cactus runtime system itself.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous referees, who provided valuable comments that substantially improved the paper. This work supported in part by the Defense Advanced Research Projects Agency under grant N66001-97-C-8518, the National Science Foundation under grants CDA-9500991 and ANI-9979438, the Department of Energy Office of Science grant DE-FG02-05ER25662, and the Sandia University Research Program contract 190576.

REFERENCES

[1] J. Postel, "Transmission control protocol," RFC 793, 1981.

[2] ——, "User datagram protocol," RFC 768, 1980.

[3] S. Kent and R. Atkinson, "Security architecture for the internet protocol," RFC (Standards Track) 2401, 1998.

[4] A. O. Freier, P. Karlton, and P. Kocher, "The SSL protocol, version 3.0," Netscape Communications," Internet-Draft, 1996.

[5] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala, "RSVP: A new resource ReSerVation Protocol," *IEEE Network*, 1993.

[6] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "RTP: A transport protocol for real-time applications," RFC 1889, 1996.

[7] R. Wu and A. Chien, "GTP: Group transport protocol for lambda-grids," in *IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2004)*. IEEE Press, 2004. [Online]. Available: http://vgrads.rice.edu/papers/vgrads5/attach/GTP_CCGrid2004.pdf

[8] E. Weigle and A. A. Chien, "The composite endpoint protocol (CEP): Scalable endpoints for terabit flows," in *Proceedings of the IEEE Conference on Cluster Computing and the Grid (CCGrid 2005)*, 2005.

[9] R. R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. J. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson, "Stream control transmission protocol," Internet Draft, 2000.

[10] M. A. Hiltunen, R. D. Schlichting, X. Han, M. Cardozo, and R. Das, "Real-time dependable channels: Customizing QoS attributes for distributed systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 6, pp. 600–612, 1999.

[11] *Data Signature Standard (DSS)*. National Institute of Standards and Technology, FIPS PUB 186-2, U.S. Department of Commerce, Washington, D.C., 2000.

[12] N. Hutchinson and L. L. Peterson, "The *x*-kernel: An architecture for implementing network protocols," *IEEE Transactions on Software Engineering*, vol. 17, no. 1, pp. 64–76, 1991.

[13] M. A. Hiltunen, "Configuration management for highly-customizable software," *IEE Proceedings: Software*, vol. 145, no. 5, pp. 180–188, 1998.

[14] W.-K. Chen, M. A. Hiltunen, and R. D. Schlichting, "Constructing adaptive software in distributed systems," in *Proceedings of the 21st International Conference on Distributed Computing Systems*, Mesa, AZ, 2001, pp. 635–643.

[15] J. Appavoo, K. Hui, C. A. N. Soules, R. W. Wisniewski, D. D. Silva, O. Krieger, M. Auslander, D. Edelsohn, B. Gamsa, G. R. Ganger, P. McKenney, M. Ostrowski, B. Rosenburg, M. Stumm, and J. Xenidis, "Enabling autonomic system software with hot-swapping," *IBM Systems Journal*, vol. 42, no. 1, pp. 60–76, 2003.

[16] L. Rizzo, "Effective erasure codes for reliable computer communication protocols," *Computer Communication Review*, vol. 27, no. 2, pp. 24–36, 1997.

[17] V. Jacobson, "Congestion avoidance and control," in *Proceedings of ACM SIGCOMM '88*, 1988, pp. 314–332.

[18] S. Floyd, M. Handley, J. Padhye, and J. Widmer, "Equation-based congestion control for unicast applications," in *Proceedings of ACM SIGCOMM 2000 Symposium*, 2000.

[19] S. Cen, C. Pu, and J. Walpole, "Flow and congestion control for Internet streaming applications," in *Proceedings of Multimedia Computing and Networking 1998*, 1998.

[20] K. K. Ramakrishnan and S. Floyd, "A proposal to add Explicit Congestion Notification (ECN) to IP," RFC 2481, 1999.

[21] P. G. Bridges, "Composing and coordinating adaptation in cholla," Ph.D. dissertation, University of Arizona, Tucson, Arizona, 2002.

[22] M. Rajagopalan, S. K. Debray, M. A. Hiltunen, and R. D. Schlichting, "Profile-directed optimization of event-based programs," in *Proceeding of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. ACM Press, 2002, pp. 106–116.

[23] C. Chambers, S. Eggers, J. Auslander, M. Philipose, M. Mock, and P. Pardyak., "Automatic dynamic compilation support for event dispatching in extensible systems," in *Proceedings of the 1996 Workshop on Compiler Support for Systems Software (WCSSS-96)*, February, 1996.

[24] R. Wu, A. Chien, M. Hiltunen, R. Schlichting, and S. Sen, "A high performance configurable transport protocol for Grid computing," in *Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2005)*, Cardiff, Wales, 2005.

[25] X. Forum, "Xpress transport protocol specification, revision 4.0," XTP Forum, 1394 Greenworth Place, Santa Barbara, CA 93108, March 1995.

[26] D. C. Feldmeier, "An overview of the TP++ transport protocol project," *High Performance Networks, Frontiers and Experience*, pp. 157–176, 1994.

[27] G. J. Minden, E. Komp, S. Ganje, M. Kannan, S. Subramaniam, S. Tan, S. Vallabhaneni, and J. B. Evans, "Composite protocols for innovative active services." in *Proceedings of the 2002 DARPA Active Networks Conference and Exposition (DANCE 2002)*, 2002, p. 157ff.

[28] D. M. Ritchie, "A stream input-output system," *AT&T Bell Laboratories Technical Journal*, vol. 63, no. 8, pp. 311–324, 1984.

[29] F. Travostino, E. M. III, and F. Reynolds, "Paths: Programming with system resources in support of real-time distributed applications," in *Proceedings of the IEEE Workshop on Object-Oriented Real-Time Dependable Systems*, 1996.

[30] R. van Renesse, K. P. Birman, R. Friedman, M. Hayden, and D. A. Karr, "A framework for protocol composition in Horus," in *Proceedings of the 14th ACM Principles of Distributed Computing Conference*, 1995, pp. 80–89.

[31] R. van Renesse, K. P. Birman, M. Hayden, A. Vaysburd, and D. A. Karr, "Building adaptive systems using Ensemble," *Software Practice and Experience*, vol. 28, no. 9, pp. 963–979, 1998.

[32] W. Allcock, J. Bresnahan, R. Kettimuthu, and J. M. Link, "The Globus eXtensible Input/Output system (XIO): A protocol independent IO system for the Grid," in *Proceedings of the 2003 International Parallal and Distributed Processing Symposium (IPDPS)*, 2005.

[33] G. Parr and K. Curran, "A paradigm shift in the distribution of multimedia," *Communications of the ACM*, vol. 43, no. 6, pp. 103–109, 2000.

[34] R. Braden, T. Faber, and M. Handley, "From protocol stack to protocol heap — role-based architecture," in *Proceedings of the First Workshop on Hot Topics in Networks (HotNets-I)*, 2002. [Online]. Available: citeseer.nj.nec.com/braden02from.html

[35] N. Bhatti, M. A. Hiltunen, R. D. Schlichting, and W. Chiu, "Coyote: A system for constructing fine-grain configurable communication services," *ACM Transactions on Computer Systems*, vol. 16, no. 4, pp. 321–366, 1998.

[36] D. C. Schmidt, D. F. Box, and T. Suda, "ADAPTIVE: A dynamically assembled protocol transformation, integration, and evaluation environment," *Concurrency: Practice and Experience*, vol. 5, no. 4, pp. 269–286, 1993.

[37] P. Patel, A. Whitaker, D. Wetherall, J. Lepreau, and T. Stack, "Upgrading transport protocols using untrusted mobile code," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, 2003.
[38] D. Velten, R. Hinden, and J. Sax, "Reliable data protocol," RFC 908, 1984.
[39] C. Partridge and R. Hinden, "Version 2 of the reliable data protocol," RFC 1151, 1990.
[40] D. Cheriton, "VMTP: Versatile message transaction protocol," RFC 1045, 1988.
[41] R. Marasli, P. D. Amer, and P. T. Conrad, "Partially reliable transport service," *Proceedings of the 2nd IEEE Symposium on Computers and Communications (ISCC)*, 1997. [Online]. Available: citeseer.nj.nec.com/87031.html
[42] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "TCP selective acknowledgment options," RFC 2081, 1996.
[43] R. Braden, "T/TCP – TCP extensions for transactions," RFC 1644, 1994.

PLACE PHOTO HERE

Patrick G. Bridges is an assistant professor with the Computer Science Department at the University of New Mexico. He received his Ph.D. from the University of Arizona and his B.S. from Mississippi State University. Bridges is a member of the ACM and the IEEE Computer Society. His research interests include high-performance computing systems, configurable and adaptable system software, and system-wide measurement and monitoring techniques.

PLACE PHOTO HERE

Gary T. Wong is a researcher with Boston University's Computer Science Department, where he collaborates with many others on a video sensor network project. His past work covers a range of topics from compilers to distributed systems, but he is particularly interested in exploring novel structures for operating systems. He holds a Bachelor of Science degree from the University of Auckland.

PLACE PHOTO HERE

Richard D. Schlichting is currently Director of Software Systems Research at AT&T Labs-Research in Florham Park, NJ. He received the B.A. degree in mathematics and history from the College of William and Mary, and the M.S. and Ph.D. degrees in computer science from Cornell University. He was on the faculty at the University of Arizona from 1982-2000, and spent sabbaticals in Japan in 1990 at Tokyo Institute of Technology and in 1996-97 at Hitachi Central Research Lab. Schlichting is an ACM Fellow and an IEEE Fellow, and is on the editorial board of IEEE Transactions on Software Engineering. He has also the current chair of IFIP Working Group 10.4 on Dependable Computing and Fault-Tolerance, and has been active in the IEEE Computer Society Technical Committee on Fault-Tolerant Computing, serving as chair from 1998-99. His research interests include distributed systems, highly dependable computing, and networks.

PLACE PHOTO HERE

Matti A. Hiltunen is a researcher in the Dependable Distributed Computing and Communication department at AT&T Labs-Research in Florham Park, NJ. He received the M.S. degree in computer science from the University of Helsinki and the Ph.D. degree in computer science from the University of Arizona. Hiltunen is a member of the ACM, the IEEE Computer Society, and IFIP Working Group 10.4 on Dependable Computing and Fault-Tolerance. His research interests include dependable distributed systems and networks, grid computing, and pervasive computing.

PLACE PHOTO HERE

Matthew J. Barrick is a Ph.D. student in the Computer Science Department at the University of New Mexico. He currently works in the UNM Scalable Systems Lab on issues ranging in a variety of areas, including high-performance large-scale file systems and scalable multiprocessor operating systems like K42. He received his B.S. from the Indiana University of Pennsylvania, and is a member of the ACM.