A Configurable and Extensible Transport Protocol

Gary T. Wong
Department of Computer Science
The University of Arizona
Tucson, AZ 85721
gary@cs.arizona.edu

Matti A. Hiltunen and Richard D. Schlichting
AT&T Labs - Research
180 Park Avenue
Florham Park, NJ 07932
{hiltunen,rick}@research.att.com

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 an approach to building such customized protocols using Cactus, a system in which micro-protocols implementing individual attributes of transport can be combined into a composite protocol that realizes the desired overall functionality. In contrast with similar systems, Cactus supports non-hierarchical module composition and event-driven execution, both of which increase flexibility and allow finer-grain modules implementing orthogonal properties. To illustrate this approach, the design and implementation of a configurable transport protocol called CTP is presented. CTP allows customization of a number of properties including reliable transmission, congestion detection and control, jitter control, and message ordering. This suite of micro-protocols has been implemented using Cactus/C 2.0 on Red Hat Linux 6.2, with initial experimental results indicating that the ability to target the guarantees more precisely to the needs of applications can in fact result in better performance.**

Keywords— **Transport protocols, configuration, customization, protocol design, extensibility, composibility, reuse.**

I. Introduction

EXISTING network transport protocols such as TCP [1] and UDP [2] have limitations when they are utilized 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 overreact 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, and SCTP [7] for enhanced reliability and delivery over independent parallel streams. Developing such a protocol from scratch is, needless to say, often a significant undertaking.

In this paper, we argue that building customized transport protocols from collections of fine-grain modules is a viable and effective alternative to relying on existing or specialized protocols. With this approach, modules are chosen 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.

This work supported in part by the Defense Advanced Research Projects Agency under grant N66001-97-C-8518 and the National Science Foundation under grants ANI-9979438 and CDA-9500991.

We substantiate our arguments by describing the design and prototype implementation of CTP, a configurable transport protocol that represents a concrete realization of this approach. CTP is built using Cactus, a design and implementation framework for constructing highly-configurable network services [8]. In Cactus, each service attribute or variant is implemented as an independent software module called a *micro-protocol*. Micro-protocols are structured using an event-driven execution paradigm and can share data. A customized version of the service is constructed by choosing micro-protocols based on the desired properties and linking them together with a runtime system to give a *composite protocol*, which is then composed hierarchically with other composite protocols and standard protocols to form the network subsystem. When compared with similar systems for building configurable protocols [9], [10], [11], [12], Cactus provides finer granularity, a two-level composition model with both hierarchical and non-hierarchical composition, and a flexible and dynamic event mechanism that maximizes the configurability of micro-protocols.

Several prototype implementations of Cactus have been constructed, including one written in C that runs on Linux and the MK Mach OS from OpenGroup [13], another written in C++ that runs on Solaris and Linux, and a third written in Java that runs on multiple platforms. CTP is being implemented using the C version of Cactus on a cluster of Pentiums running Red Hat Linux version 6.2. Other prototype services that have been successfully implemented using Cactus or the predecessor Coyote system [14] include group RPC [15], membership [16], and a real-time channel abstraction [8].

This paper has several goals. The first is to argue that fine-grain configurability and extensibility are valuable characteristics for transport protocols. The second is to describe a realization of this philosophy in the form of CTP. The last is to present initial performance results, which illustrate that the cost of such flexibility is relatively small and that the advantages of configurability can more than compensate for this overhead in certain cases.

II. CTP Design

Each service attribute and functional component of CTP is designed to be realized as a separate module in such a way that the modules can be combined to provide a transport protocol with exactly the required attributes. This section describes the design of CTP, starting with an overview of Cactus.

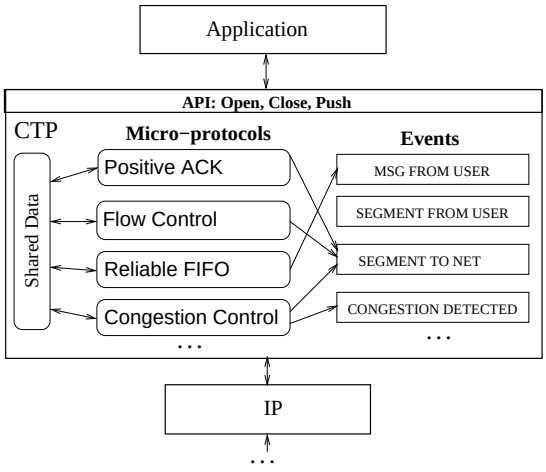

Fig. 1. CTP Composite Protocol in Cactus.

A. Cactus

Cactus is a system and a framework for constructing configurable protocols and services, where each service property or functional component is implemented as a separate module [8]. A service in Cactus is implemented as a *composite protocol*, with each service property or other functional component implemented as a *micro-protocol*. A micro-protocol is, in turn, structured as a collection of *event handlers*, which are procedure-like segments of code that are executed when a specified event occurs. Once constructed, a composite protocol is composed hierarchically with other protocols to form the network subsystem. In the case of the Linux version of Cactus used to implement CTP, support for hierarchical composition is provided by the *x*-kernel [9].

The Cactus runtime system provides a variety of operations for managing events and event handlers. In particular, operations are provided for binding an event handler to a specified event and for raising an event, which causes all the handlers bound to that event to be executed. An event can also be raised with a specified delay to implement time-driven execution, and with either blocking or non-blocking semantics on the thread raising the event. The order of event handler execution can also be specified if desired. Arguments can be passed to handlers in both the bind and raise operations. 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. Cactus also supports shared data that can be accessed by all micro-protocols configured into a composite protocol. Fig. 1 illustrates CTP implemented as a Cactus composite protocol, with example events to the right and micro-protocols to the left. An arrow from a micro-protocol to an event indicates that the micro-protocol binds a handler to the event.

Finally, Cactus supports a message abstraction designed to facilitate development of configurable services. The main features provided by Cactus messages are named *message attributes* and a coordination mechanism that releases a message from the composite protocol to go up or down the protocol graph only when agreed to by all relevant micro-protocols. These dynamically created message attributes are a generalization of traditional message headers and have three different scopes: *peer*, *stack*, and *local*. Peer attributes correspond to traditional header fields that are shared by the peer composite protocols at the sender and receiver. Stack attributes are shared by different protocol layers in a protocol stack on one machine and can be used, for example, to share message-specific processing instructions between protocol layers. Finally, local attributes are shared by micro-protocols in one composite protocol on a single machine and can be used, for example, to store message-specific local information. A customizable pack routine combines peer attributes with the message body for network transmission, while an analogous unpack routine extracts peer attributes at the receiver. Messages are deallocated using a coordination mechanism similar to that used for sending messages.

B. Attributes and Algorithms

The first step in developing a customizable transport protocol is to identify various quality attributes that can be provided to higher levels and the algorithms used to implement these and other aspects of the service. While the list of possible quality attributes is large [17], the ones we address here can be divided roughly into the following categories:

- *Performance.* Describes how quickly data are transported from sender to receiver, typically specified as 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 using appropriate algorithms.
- *Reliability.* Addresses the probability that the receiver receives all the data sent by the sender. Reliability can be increased by using different forms of redundancy ranging from multihoming to redundant transmission of data along one connection. Since most techniques transmit multiple copies, the transport protocol may be required to eliminate extra copies.
- *Ordering.* Describes guarantees concerning the ordering of data at the receiver relative to the order in which they were sent. For stream-based transport services, the only reasonable ordering option is FIFO, but for message-based services other options may be reasonable.
- *Security.* Addresses confidentiality, integrity, authenticity, and data replay. The strength of the guarantee for each of these attributes depends on the types of attacks to be tolerated.

In general, the chosen quality attributes apply to every message within a session, but it may also be useful to allow individual messages to be given particular attributes. For example, an urgent message might be marked “out-of-band” and delivered as soon as possible, even though a message ordering requirement applies to other messages. Applications that require multi-

ple substreams with different characteristics, such as multimedia delivery of parallel audio/video channels, can open multiple sessions with different quality attributes and distribute traffic across these sessions as appropriate.

TCP and UDP provide essentially a fixed set of these attributes. In particular, TCP provides a stream-based reliable ordered transmission with integrity against accidental data modification, but only best-effort performance and with no timeliness or security guarantees. UDP provides a best-effort message-based transport with no 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 ones complement and cyclic redundancy checks (CRC) provide integrity that protects against accidental data modification, while cryptographic methods such as keyed MD5 [18] protect against intentional modification. In other cases, different algorithms provide approximately the same guarantee, but with different tradeoffs with respect to resource usage or other attributes. For example, forward error correction typically uses more bandwidth than acknowledgments, but provides faster recovery from failures, and thus 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 graph. 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 teardown.

C. Design Overview

In CTP, each attribute or function described above is implemented by one micro-protocol or a set of alternative micro-protocols. Thus, the current design has one or more micro-protocols for reliability, ordering, security, jitter control, congestion detection and control, flow control, data and header compression, MTU discovery, message fragmentation and collation, and connection establishment, monitoring and teardown. The current prototype supports Application Layer Framing [19], so applications can associate specific semantics with each message. All transmission properties are defined in terms of these messages, which can be of arbitrary size. CTP can fragment or coalesce the messages as needed into appropriate transport units—*segments*—for the lower-level protocol.

The goal of the design is to decouple the implementation of different attributes and functions to maximize the ability to mix and match different micro-protocols 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 share 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.

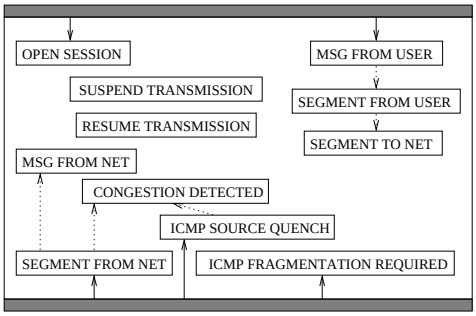

Fig. 2. Major CTP events.

Micro-protocols interact using shared data—in particular, the messages and their attributes—and the set of events illustrated in Fig. 2. The figure uses solid arrows to indicate events raised by the boundary layers of the CTP protocol and dashed arrows to indicate causal relations between other events. For example, when the MSG FROM USER event is raised by CTP, some micro-protocol will raise the SEGMENT FROM USER event. Additional local timeout events are used by several of the micro-protocols. Most of the event names are self explanatory. The ICMP events are used to process control messages transmitted by the Internet Control Message Protocol to higher levels, such as those indicating congestion (ICMP SOURCE QUENCH) or that a packet was dropped because it was too large (ICMP FRAGMENTATION REQUIRED). The SUSPEND TRANSMISSION and RESUME TRANSMISSION events are used to control the number of messages that the application can push to the CTP service at the sender. In particular, any micro-protocol that needs to control the rate at the application for flow or congestion control can raise these events, which are monitored by micro-protocol Application Control. The latter then regulates the application when necessary by, for example, blocking the application thread on a semaphore. The CONGESTION DETECTED event allows the congestion detection mechanism to be separated from the actual congestion control, thereby making it easy to change the detection method if desired.

CTP also takes advantage of the transmission coordination aspect of Cactus messages, both for sending segments down to lower layers and for delivering messages up to the application. This coordination is implemented by *send bits* that are associated with each message. Each micro-protocol that processes a given message type is allocated a send bit in each message of that type that it sets when it is done processing the message. The message then exits the composite protocol either up or down as appropriate once all send bits have been set. For example, congestion control, flow control, and reliability all use send bits to determine when a segment can be transmitted, while jitter control and the different ordering micro-protocols use send bits to determine when a message can be delivered to the application.

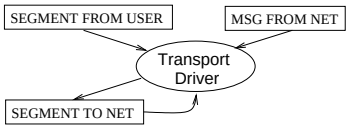

Fig. 3. Transport Driver event handling.

Send bits allow different micro-protocols to operate on messages independently without knowing which other micro-protocols need to process the message. They also decouple the approval process from any kind of ordering—when all the required micro-protocols have set their bit, 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.

III. Micro-protocol Highlights

This section gives an overview of some of the micro-protocols available in the CTP suite, including those that implement reliable delivery, transmission control, message ordering, and jitter. There are also several micro-protocols that provide base functionality not directly connected with a specific semantic property. Finally, the section concludes with a discussion of the configurability and extensibility of the service.

A. Base Functionality

Transport Driver is the only micro-protocol 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 micro-protocols. It also sets the send bits to ensure that messages are sent even if there are no other micro-protocols that set send bits in the configuration. The event interactions of Transport Driver are illustrated in Fig. 3. In the figure, arrows pointing to a micro-protocol indicate that the micro-protocol has a handler bound to the event and arrows originating in the micro-protocol indicate that the micro-protocol raises the event. Pseudo-code for this micro-protocol is given in Fig. 4.

The Sequenced Messages and Sequenced Segments micro-protocols 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 micro-protocols such as those providing reliability and ordering. Performing the procedure in a separate micro-protocol allows the other micro-protocols to share the same attribute, saving space in the message.

A group of micro-protocols 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, Coalescence combines multiple small messages into one segment, and Resizing fragments the messages into segments that can be handled by the underlying IP network without IP-level fragmentation (MTU discovery). One of these micro-protocols must be present in each configuration.

```
micro-protocol Transport Driver () {

  handler handleUsrSeg(segment s){
    raise (SEGMENT TO NET, SYNC, 0, s);
  }
  handler handleSegToNet(segment s){
    /* Set the demultiplexing keys for the peer */
    setAttr( s, PEER, SrcPort, sessn→portLocal );
    setAttr( s, PEER, DestPort, sessn→portPeer );
    setSendBit (s); setDeallocateBit (s);
  }
  handler handleNetMsg(message m){
    setSendBit (m); setDeallocateBit (m);
  }
  initial {
    bind(SEGMENT FROM USER,handleUsrSeg,1024);
    bind(SEGMENT TO NET,handleSegToNet,0);
    bind(MSG FROM NET,handleNetMsg,0);
  }
}
```

Fig. 4. Transport Driver micro-protocol pseudo code

Finally, a set of optional micro-protocols 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 micro-protocols, 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, the micro-protocol 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 micro-protocols 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 micro-protocol is responsible for sending probe messages to detect link failures in the absence of application messages.

B. Reliability Micro-protocols

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 micro-protocols: Positive ACK and FEC Reliability. Positive ACK is a traditional ARQ reliability scheme, but differs in implementation details from the one in TCP because it attempts to decouple reliability processing from other communication properties. For instance, Positive ACK places no limit on the number of outstanding unacknowledged segments; if a limit on outgoing segments is required, the user must configure a flow control micro-protocol. FEC Reliability uses the forward error correction algorithm in [20] to transmit redundant data so that the receivers can reconstruct a complete transmission despite message losses.

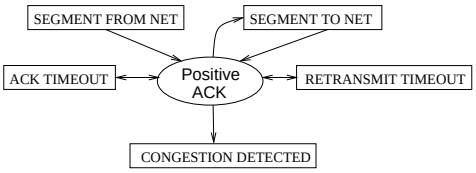

Fig. 5. Positive ACK event handling.

Positive ACK is a relatively complex micro-protocol, as illustrated in Fig. 5. For each outgoing message (event SEGMENT TO NET), it includes an acknowledgment attribute indicating the most recent segment received in order and raises the RETRANSMIT TIMEOUT timer event. For each incoming messages (event SEGMENT FROM NET), it checks the acknowledgment attribute and deallocates any segments that it now knows have been received and cancels the retransmission event if appropriate. If the incoming message contains data, i.e., is not just an acknowledgment, Positive ACK updates its record of the last segment received and raises the ACK TIMEOUT timer event. When the retransmission event occurs, a check is made of how many times the segment has been transmitted and if the number is excessive, the connection is aborted. Otherwise, the retransmission delay is increased using exponential backoff, and the SEGMENT TO NET event is raised on the oldest unacknowledged segment. The ACK TIMEOUT event handler merely creates an acknowledgment message and then raises the SEGMENT TO NET event. The congestion event is raised when a retransmission occurs, because it can provide useful information to a congestion control micro-protocol when one is included in the configuration.

Note that several optimizations commonly made in ARQ schemes are omitted here, such as fast ACK transmission and duplicate ACK detection. These could easily be added to CTP. Some of these techniques would be easier to structure as new micro-protocols, while others would be more natural to provide as extensions to existing micro-protocols.

The FEC Reliability micro-protocol encodes k segments of original data into n segments of encoded data ($n > k$) using the algorithm in [20]. 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. FEC Reliability at the receiver then intercepts the redundant segments and uses them to create a new message for each of these missing segments and raises the SEGMENT FROM NET event. As a result, other micro-protocols see the reconstructed segments as if they had arrived normally.

By combining forward error correction and some kind of ARQ reliability scheme, the user can choose from a rich set of possibilities for reliable communication that can be used to match the specific requirements of particular applications.

C. Transmission Control Micro-protocols

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 micro-protocols that implement these functions into three categories: flow control, congestion control, and application 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 micro-protocols include:

- *XON/XOFF.* The receiver issues suspend/resume instructions to the sender.
- *Windowed.* The receiver periodically informs the sender of its available buffer space.
- *Rate based.* The sender paces its transmission according to some limit on traffic per time period; the limit may be fixed or may vary based on feedback from the receiver.

These micro-protocols 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. This event handler is also capable of detecting when further messages cannot be sent immediately; if this is the case, it raises a SUSPEND TRANSMISSION event and eventually a RESUME TRANSMISSION event once traffic can resume. These events can be handled by the application control micro-protocols described below.

At the receiver side, there are facilities in the API to allow higher level protocols to specify policies on traffic rates. The flow-control micro-protocols 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.

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 micro-protocols: congestion detection and congestion control. Typical configurations would include one congestion control and one or more congestion detection micro-protocols.

All congestion detection micro-protocols raise the CONGESTION DETECTED event when they suspect congestion in the network. In the current design, congestion control micro-protocols simply use the presence of this event to throttle transmission, but it would be possible to add voting or other processing on the congestion events before any action is made.

Available congestion detection micro-protocols include:

- *Retransmission detection.* Any reliability micro-protocol can also function as a congestion detection micro-protocol since the loss of a message may be due to congestion in the network. In the case of retransmission micro-protocols, the sender raises the congestion event when it retransmits a message.
- *Loss notification.* Even if retransmissions are not required, receivers can assist in congestion detection by transmitting feedback about losses in the incoming traffic. This can be useful for detecting congestion in unreliable communication, or in combination with FEC Reliability.
- *Roundtrip time measurement.* Large increases in roundtrip times can be an indication of queuing delays caused by buffering

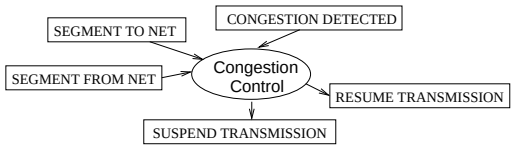

Fig. 6. `Congestion Control` event handling.

in the network. A congestion detection micro-protocol can measure end-to-end roundtrip times and raise the congestion event when appropriate.

- *ICMP detection.* In IP networks, an ICMP Source Quench message is defined to allow gateways to notify hosts about congestion. An ICMP detection micro-protocol should raise a congestion event upon receiving a Source Quench.
- *ECN detection.* If gateways provide Explicit Congestion Notification (ECN) as proposed in [21], a micro-protocol at the receiver can feed this information back to the sender as a means for detecting congestion.

Congestion control micro-protocols can use similar mechanisms to the flow control micro-protocols described above to establish an appropriate transmission rate, but must base their rates on the presence of CONGESTION DETECTED events rather than feedback from receivers. Both rate-based and window-based congestion control are possible, and these micro-protocols have considerable scope for flexibility in tuning their transmission to maximize throughput while avoiding congestion. Fig. 6 illustrates typical event interactions of a congestion control micro-protocol. Note that a rate-based micro-protocol would use an additional timer event.

Application control micro-protocols can be configured into CTP to provide whatever properties are desired when outgoing data cannot be transmitted immediately. With no such micro-protocol, messages simply queue up within the CTP composite protocol at the sender until all send bits are set; there is no upper bound on the messages that will be buffered by default. It is possible to provide such a limit by keeping a count of the messages allocated and deallocated by the protocol, or the size of all messages currently allocated. The application control micro-protocol can decide on appropriate behavior based on the currently queued data and whether immediate transmission is possible by counting SUSPEND TRANSMISSION and RESUME TRANSMISSION events. It registers an event handler for MSG FROM USER events to enforce its policy, which might include:

- *Blocking.* Incoming application threads are blocked at a FIFO semaphore until their messages can be accepted by CTP. This option gives semantics similar to blocking I/O in BSD sockets.
- *Error report.* The protocol notifies the application that a message cannot yet be accepted for delivery. This option gives semantics similar to non-blocking I/O in BSD sockets.
- *Dropping.* Incoming excess messages are dropped. In certain real-time applications, late arriving data may have little or no value to the receiver, and thus it may be better for the protocol to discard additional messages immediately, rather than buffer them and introduce further queuing delays.

D. Ordering and Jitter Micro-protocols

Ordering micro-protocols are relatively simple for point-to-point communication such 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 two FIFO micro-protocols, Reliable FIFO and Lossy FIFO. The former enforces strict in-order delivery by buffering out-of-order messages and sending them to the application only after their predecessors have been delivered, while Lossy FIFO delivers messages immediately, discarding any messages that arrive out of order. A Semantic Order micro-protocol uses ordering information provided by the application to record and enforce the logical predecessors of each message. An Out of Band micro-protocol can be used with any ordering micro-protocol to allow urgent out-of-band messages to be delivered as quickly as possible. This micro-protocol simply overrides the send bit used by the current ordering micro-protocol.

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

E. Discussion: Configurability and Extensibility in CTP

To make CTP highly configurable, the different micro-protocols have been designed to be as independent as possible. However, there are some *dependencies*—when one micro-protocol requires that another be in the configuration to function correctly—and some *conflicts*—when two micro-protocols 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 micro-protocols Fixed Size, Coalescence, or Resizing. The reliability and FIFO ordering micro-protocols use sequence numbers provided by Sequenced Segments and Sequenced Messages, respectively. Micro-protocols such as flow control and congestion control that may need to control application sending speed require one application control micro-protocol in each configuration. Finally, congestion control requires at least one congestion detection micro-protocol. Note that the reliability, ordering, and flow control transport functions that are often tightly connected in TCP implementations are completely independent in our design.

Conflicts are either syntactic or semantic in nature. An example of a syntactic conflict is that only one message-to-segment conversion micro-protocol should be in each configuration, while an example of a semantic conflict is that Lossy FIFO and a reliable communication micro-protocol 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 micro-protocols. The other syntactic conflicts are that there must be at most one application control, ordering, and jitter control micro-protocol in each configuration.

Despite these dependencies and conflicts, there are still hundreds of possible different CTP configurations even with a small number of different micro-protocols 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. A graphical tool called the CactusBuilder is provided to help automate the construction of composite protocols for this purpose [22].

CTP is also designed to be easily extensible, meaning that new micro-protocols 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 interactions are usually the same as in existing micro-protocols. For example, a reliability micro-protocol based on negative acknowledgments would interact with other micro-protocols in a manner identical to Positive ACK.

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 micro-protocol. If not, the CTP framework or some of the existing micro-protocols may need to be modified to raise these events. For example, for a more elaborate congestion detection micro-protocol that monitors retransmissions, retransmission micro-protocols would be modified to raise a retransmission event rather than directly raising CONGESTION DETECTED. However, completely new micro-protocols can often be implemented using the existing set of events. For example, in our design, the jitter control micro-protocols were added after the rest of CTP was designed with no modifications to other micro-protocols.

IV. Initial Performance Results

A. Overview

This section gives some initial CTP performance results that indicate both the overhead and the potential advantages of configurability. 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 have not been considered in the design 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 than TCP are needed. CTP is also appropriate when there is the opportunity to configure a protocol to match the characteristics of a specific network environment.

B. Simple Transmission Test

The first set of experiments measures the time taken to transmit 10 large Ethernet frames using UDP, TCP and various configurations of CTP. In these tests, a sending application transmits enough data to fill 10 frames to a receiver, which replies with an acknowledgment once all the data have been received. We measure the interval at the sender between the transmission of the first frame and receipt of the acknowledgment. The application sends 1400 bytes at a time to leave sufficient room for lower level headers in a single Ethernet frame of 1500 bytes. UDP and CTP deliver this data in a single frame, while TCP attempts to send as much data as possible in each outgoing segment. In each case, exactly 10 Ethernet frames are required. The test deliberately limits the data transferred to this relatively small amount to avoid overrunning buffers in the network and endpoints; UDP and configurations of CTP without flow or congestion control cannot reliably deliver larger amounts of data under such conditions, so a direct comparison across all protocols would be unfair.

TABLE I
DELAY COMPARISON

Protocol	Time (ms)
UDP	2.58
TCP	3.04
CTP	3.79
CTP: Lossy FIFO	4.03
CTP: Positive ACK	4.40
CTP: Positive ACK and Reliable FIFO	4.52

The tests were conducted on two 300 MHz Pentium II CPUs running Linux kernel 2.2.14 across a 100 Mbps Ethernet. The TCP implementation is a port of the 4.3 BSD Reno protocol modified to run as a user-space *x*-kernel protocol. The TCP measurements were made with the flow control and congestion windows fully open, and the PUSH flag was set on the last segment to ensure the data were sent with no delay. Thus, every effort was made to ensure the baseline TCP measurements are as fast as possible. The times are averages from 20 tests for each case; the standard error of each figure is approximately 0.03 ms.

These preliminary results indicate an overhead of less than 1.5 ms (150 μs per message) over UDP and TCP with approximately the same service guarantees in this simple test and execution environment. We expect the performance of CTP to improve as the implementation is optimized, although it is probably unrealistic to expect CTP to beat TCP and UDP for this type of use.

C. Other Configurations

CTP can be tuned to provide optimized behavior for given applications or environments, which can give it a performance advantage in those cases. One example is providing fast retransmission if network latency is known to be low. Versions of TCP derived from the BSD code retransmit segments after receiving 3 duplicate ACKs or upon expiration of a retransmission timeout. However, the retransmission timer is typically very coarse, on the order of 500ms. Connections across campus networks or even wide area-networks frequently yield round-trip times on the order of tens of milliseconds, so faster retransmission timers can be beneficial under certain circumstances.

We have measured the performance of CTP configured with the Positive ACK micro-protocol described in section III-B us-

TABLE II
END-TO-END LATENCY COMPARISON

Loss rate (%)	CTP time (ms)	TCP time (ms)
0	0.5	0.3
1	1.1	10.9
2	2.3	18.2
3	2.4	27.8
4	3.0	46.1
5	3.3	57.6

ing fine-grained retransmission timing. The test measures the time to transmit one small message from the sender to the receiver. Each message is transmitted individually, which means that the receiver in the TCP experiment does not detect message loss based on gaps in the message stream and does not generate duplicate ACKs. As a result, the sender only retransmits when the retransmission timer expires. Table II lists the average end-to-end latency of this CTP configuration compared to TCP. This test was made on the same platform as described above; network packet losses were simulated by dropping varying proportions of packets. The system clocks were synchronized with NTP and timestamps were added by the sender and read by the receiver. The times are averages of 1000 trials, with an equal number of trials made in each direction to minimize bias introduced by discrepancies between the clocks.

Although TCP is faster 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 wireless networks. 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 micro-protocols, 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.

Another type of optimization modifies the Cactus runtime system, but does not require changes in the micro-protocol 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.

Finally, some optimizations require that the chosen micro-protocols 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. The delay for the unoptimized CTP in the tests described in section IV-B with the basic configuration was 4.67 ms, while the delay was 3.76 ms after making this optimization. This only took a few hours of programming and did not require any changes to the Cactus framework or CTP micro-protocols. Testing other types of optimizations is part of planned future work.

E. Memory Use

The current prototype of CTP contains significant testing and debugging code, and makes little attempt to minimize memory use. The static memory requirements for this implementation of the CTP consist of 24 KB for the Cactus framework, 15 KB for the interface to the *x*-kernel, 5 KB for the CTP infrastructure, and various amounts for each micro-protocol. Micro-protocol memory use varies from a few hundred bytes for simple micro-protocols like Sequenced Segments, to approximately a kilobyte for typical modules such as Reliable FIFO. Particularly complex micro-protocols like FEC Reliability may require as much as 10 KB.

In addition to these static requirements, a few micro-protocols allocate significant amounts of memory at runtime. For example, ordering micro-protocols buffer messages received out of order, retransmission micro-protocols save copies of messages at the sender in case subsequent transmissions are required, and forward error correction micro-protocols require space at both the sender and receiver for whatever messages are required for performing the redundancy calculations. These dynamic memory requirements vary enormously depending on the choice of micro-protocols, the parameters specified for those micro-protocols, and other environmental criteria such as the connection bandwidth-delay product.

Overall, we estimate that an implementation of CTP designed to minimize memory use could operate within 32 KB plus space for the desired micro-protocols and their dynamic memory requirements. While this space is excessive for the smallest embedded microcontroller environments, we believe that a careful implementation of CTP should be feasible for most platforms with modest memory constraints.

F. 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 the *x*-kernel 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 standard transport protocols, 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.

V. Related Work

Related work includes new transport protocols, proposed extensions to TCP, and research on configuration frameworks for transport protocols. In the first area, a number of other transport protocols have been proposed since the introduction of TCP and UDP. Examples include the Reliable Data Protocol (RDP) [23], which provides a message-based transport service with reliability and optional FIFO ordering guarantees, and the Versatile Message Transaction Protocol (VMTP) [24], which implements transactional RPC-style communication. The latter has certain customizable features, including optional security and customizable reliability, and some support for real time and multicast. More recent proposals include the Real-Time Transport Protocol (RTP), which supports transmission of real-time data such as audio or video over multicast network services [6], and the Stream Control Transmission Protocol (SCTP) [7], which provides improved reliability using techniques such as multihoming. A survey of transport protocols can be found in [17].

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 [25] and support for transaction-oriented services [26].

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.

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 the exchange of data and control messages between adjacent modules in the composition graph. The first system to support such composition was System V STREAMS [27], which allows a bidirectional connection between a user's process and a device or a pseudo-device called a *stream* to be extended dynamically by the addition of new modules. The *x*-kernel [9], and subsequent CORDS [10] and Scout [28] systems, support a more general directed protocol graph with explicit support for multiple logical sessions and multiplexing and demultiplexing. In the *x*-kernel, the protocol graph is statically configured at initialization time and different messages may take different paths through the graph depending on, for example, the size of the message. To our knowledge, none of these approaches have been used to construct transport services that are customizable to the same degree as CTP.

Similar composition models have been used in recent systems. For example, RWANDA dynamically builds a protocol stack by composing a linear sequence of modules chosen at runtime [29]. The work on Protocol Boosters [30] focuses on improving the performance of existing protocols by adding booster components that can be combined by nesting, resulting in a linear order of booster components. Finally, hierarchical composition models are used in the Horus [31] and Ensemble [32], [33] systems to construct configurable group communication services as stacks of protocol components. The interface between protocol components in these systems is richer than in systems such as the *x*-kernel to accommodate information transfer such as membership changes. These project have also worked on performance improvements by bypassing certain protocol components during normal operation [34].

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. The congestion detection and the transmission control events are a good example of how this flexibility can be used. Finally, modules in hierarchical models must typically act as data filters that get one chance to process a message when it traverses the communication subsystem. In Cactus, a micro-protocol can be much more sophisticated and can process a message multiple times while it is within a composite protocol. For example, a micro-protocol 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, micro-protocols do not need to be simple data filters, but can implement arbitrary logical transport properties.

Adaptive [12] 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. Finally, the Universal Transport Library [35] aims to deliver customized transport protocol functionality, but

focuses on providing an abstract interface under which one of many static protocols can be substituted, rather than composing a protocol from a set of independent modules.

VI. Conclusions

The ability to customize transport protocols can provide important flexibility when it comes to supporting new applications and new network technologies. Here, 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. More experiments are being conducted to explore these benefits further.

Ongoing work related to the Cactus architecture includes investigating tools to partially or completely automate the task of configuring a composite protocol to meet specified requirements, and adaptive protocols that change their behavior at runtime [36]. Results in these areas could be applied to make CTP more powerful and to simplify its use.

Other future work on CTP 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. Second, CTP will be extended 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

M. Degermark provided excellent feedback on this work, and P. Bridges ported the Cactus framework to run on Linux. The anonymous referees also gave many useful comments and suggestions.

References

[1] J. Postel, "Transmission control protocol," RFC 793, USC Information Sciences Institute, Marina del Ray, Calif., Sep 1981.

[2] J. Postel, "User datagram protocol," RFC 768, USC Information Sciences Institute, Marina del Ray, Calif., Aug 1980.

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

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

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

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

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

[8] M. Hiltunen, R. 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, Jun 1999.

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

[10] F. Travostino, E. Menze, 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*, Feb 1996.

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

[12] D. Schmidt, D. 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, Jun 1993.

[13] F. Reynolds, "The OSF real-time micro-kernel," Tech. Rep., OSF Research Institute, 1995.

[14] N. Bhatti, M. Hiltunen, R. 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, Nov 1998.

[15] M. Hiltunen and R. Schlichting, "Constructing a configurable group RPC service," in *Proceedings of the 15th International Conference on Distributed Computing Systems*, Vancouver, BC, Canada, May 1995, pp. 288–295.

[16] M. Hiltunen and R. Schlichting, "A configurable membership service," *IEEE Transactions on Computers*, vol. 47, no. 5, pp. 573–586, May 1998.

[17] S. Iren, P. Amer, and P. Conrad, "The transport layer: Tutorial and survey," *ACM Computing Surveys*, vol. 31, no. 4, pp. 360–405, Dec 1999.

[18] R. Rivest, "The MD5 message-digest algorithm," RFC 1321, MIT and RSA Data Security, Inc., Apr 1992.

[19] D. Clark and D. Tennenhouse, "Architectural Considerations for a New Generation of Protocols," in *SIGCOMM Symposium on Communications Architectures and Protocols*, Philadelphia, PA, Sept. 1990, ACM, pp. 200–208.

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

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

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

[23] D. Velten, R. Hinden, and J. Sax, "Reliable data protocol," RFC 908, BBN Communications Corporation, Jul 1984.

[24] D. Cheriton, "VMTP: Versatile message transaction protocol," RFC 1045, Stanford University, Feb 1988.

[25] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, "TCP selective acknowledgment options," RFC 2081, Oct 1996.

[26] R. Braden, "T/TCP – TCP extensions for transactions," RFC 1644, USC Information Sciences Institute, Marina del Ray, Calif., Jul 1994.

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

[28] D. Mosberger and L. Peterson, "Making paths explicit in the Scout operating system," in *Proceedings of OSDI'96*, Oct 1996, pp. 153–168.

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

[30] D. Feldmeier, A. McAuley, J. Smith, D. Bakin, W. Marcus, and T. Raleigh, "Protocol boosters," *IEEE Journal on Selected Areas in Comm.*, vol. SAC-16, no. 3, pp. 437–444, Apr 1998.

[31] R. van Renesse, K. Birman, and S Maffeis, "Horus, a flexible group communication system," *Communications of the ACM*, vol. 39, no. 4, pp. 76–83, Apr 1996.

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

[33] M. Hayden, "The Ensemble system," Tech. Rep. TR98-1662, Department of Computer Science, Cornell University, Jan 1998.

[34] R. van Renesse, "Masking the overhead of protocol layering," in *Proceedings of the ACM SIGCOMM '96*, Aug 1996, pp. 96–104.

[35] P. Conrad, P. Amer, M. Taube, G. Sezen, S. Iren, and A. Caro, "Testing environment for innovative transport protocols," *Proc. MILCOM '98*, Oct. 1998.

[36] W.-K. Chen, M. Hiltunen, and R. Schlichting, "Constructing adaptive software in distributed systems," in *Proceedings of the 21st International Conference on Distributed Computing Systems*, Phoenix, AZ, Apr 2001.