Conductor: Distributed Adaptation for Complex Networks[¤]

Mark Yarvis, An-I A. Wang, Alexey Rudenko, Peter Reiher, and Gerald J. Popek
{yarvis, awang, arudenko, reiher, popek}@ fmg.cs.ucla.edu

University of California, Los Angeles
Tech Report: CSD-TR-990042

Abstract

Computer networks are becoming more complex and diverse. Increasingly, an end-to-end connection will traverse several links with orders of magnitude differences in characteristics such as bandwidth, latency, error rate, jitter, and monetary cost. At the same time, most applications assume a level of network characteristics below which they either provide no service, or service at a cost higher than the user is willing to pay.

Conductor is an adaptation framework that moves the responsibility for network complexity out of the application and into the network. Conductor allows an application's use of the network to be tailored to the needs of the user in a manner that is transparent to the application. Conductor allows adaptation to be distributed to the points in the network where it is required. Further, Conductor allows arbitrary adaptations without compromising reliability. Finally, Conductor requires minimal changes to existing systems. All of these characteristics are incorporated into a single, integrated framework.

Conductor has been shown to be effective in important classes of problems in mobile computing, where network complexity is most prevalent today. This paper describes the Conductor architecture and presents experimental results indicating that transparent, distributed adaptation can significantly improve the user's experience in complex networks.

1 Introduction

Network applications often have major problems when network characteristics change during use. Applications must adapt to these changes to provide satisfactory service. An application's communications may cross landlines, satellite links, dial-up service, wireless relays, and asymmetric links, often without user or application awareness. These various links differ in many ways, and even a single type of link can vary substantially, causing severe problems. Early attempts at Internet telephony were largely rejected due to frequent unexplained segments of garbled content. Some users abandoned the Web because of unpredictable and intolerable delays in accessing graphically dense, but information light, content. There are many other examples.

Researchers are already aware of the effects of wide variations in bandwidth, delay, underlying reliability, noisiness, jitter, cost, security and other factors. Several solutions to particular variation problems have shown great benefit [Cohen98] [Balakrishnan95]. But these solutions handle only parts of the general problem, and that problem is likely to worsen soon. Mobility will cause more computers to interact with more kinds of networks, often under difficult conditions. Paths combining wired links and wireless links are likely to be as common tomorrow as cellular telephony for audio conversations is today. Such combined paths will cause further complications.

Network applications can adapt their behavior to these changing conditions in many ways, while still delivering the necessary services: reducing the fidelity of the transmission of video, graphics and audio; switching to text only; adding application-level redundancy for critical values; encrypting sensitive content; etc. An obvious way to add such adaptivity is to change the application itself.

However, changing the application can be complex. First, the application must be able to recognize relevant network changes. Then it must perform the appropriate adaptation, preferably without disrupting its ongoing operation. Applications can usually only perform adaptations at their endpoints, rather than within the network, since they generally treat the network as a black box. But some network problems are best solved in the network, close to the problem.

Changing the application to handle network variations may be unreasonable. It adds to the cost and complexity of application development, since it requires programmer expertise in networking, which may be unrelated to application expertise. Further, legacy applications may require re-working to run in complex networks. Finally, using a newly developed adaptation (e.g., a jitter-smoothing function) would require retrofitting many applications.

[¤] This work was partially supported by the Defense Advanced Research Projects Agency under contract DABT63-94-C-0080.

Choosing the location of adaptations is also difficult. For important simple cases, a single locus of adaptation is sufficient [Fox97]. If the data transmission only encounters one troublesome link, adaptation around that link is sufficient. But as networks become more complex, more links in the transmission will cause problems, requiring solutions that deploy and coordinate adaptations at multiple locations.

Many useful adaptations fundamentally alter the data content delivered to the end application. Dropping frames from a video, converting color images to black-and-white, and delivering text in the place of voice are some examples. This style of adaptation is challenging, since the most popular reliable transport protocol, TCP, assumes that all bits that are sent will be received. Some adaptation mechanisms avoid this problem by not supporting TCP. Others do not allow these types of adaptation. Still others introduce mediating agents to trick TCP, unfortunately introducing new points of failure to do so.

A comprehensive solution to network adaptation should possess these characteristics:

- The content of the information being exchanged must be considered [Noble97]. For example, dropping color information can reduce the bandwidth requirement of a video transmission. Doing so requires recognizing that the data type is video in a particular encoding format. The adaptors chosen for this transmission must be selected carefully, or the result will be undesirable, perhaps destroying content. Fortunately, most data exchanged in networks today is strongly typed, self describing, and easily identified. Such data can be adapted outside the application.
- Multi-machine adaptation must be coordinated. Many adaptations require that multiple machines participate. Compression is a simple example. If a compressor is deployed at one end, a decompressor must be employed at the other.
- Deciding which adaptors to execute for a given occurrence of a given application, in what order, and on what sites in the network can be difficult. The system must collect environment information, create a plan, and make good decisions, all at a low cost.
- Reliability becomes tricky. If data is being intentionally altered, what services should the communications facility provide? Guaranteed delivery of all bits submitted to the pipe is unsuitable, since the adaptation facility may intentionally drop some bits. In particular, the services of TCP are not sufficient.
- A general framework for adaptor interaction is required. Multiple adaptor groups need to work together, both nested and sequentially; late binding is essential; results are order-dependent; and independently developed adaptors must be incorporated.

Conductor is an application-level framework that demonstrates these characteristics. Conductor dynamically deploys multiple adaptors to improve an application's communication paths. Adaptors can be nested, deployed serially, or both. The framework is robust to adaptor failures. It uses *semantic segmentation* to repair such failures without resetting the channel. Conductor uses a planning algorithm to decide which adaptors to deploy.

Early experience with Conductor suggests that it works well. Measurement and observation indicate substantial improvements in the user's experience. Our research provides considerable evidence that general, application-external, dynamically negotiated communications adaptation involving multiple, cooperating locations is a promising approach.

Below, Section 2 first summarizes existing approaches to providing adaptivity in networks. Section 3 describes the design and implementation of Conductor. Section 4 gives performance results, and Section 5 discusses how widely applicable Conductor might be. Section 6 discusses ongoing and future work. Section 7 concludes.

2 Related Work

The general problem of adapting data flows for varying network conditions has been studied for several years, and has led to the development of several successful systems. These systems demonstrate that the general approach of adapting data sent over varying networks can produce acceptable results at reasonable costs. Below we outline their contributions and discuss their influence on our work.

2.1 Proxies

One simple and powerful method of providing adaptation for varying networks is to create a proxy site to assist in the use of such networks. This approach is especially applicable to mobile computers using wireless networks. Generally, a proxy is a well-connected, powerful computer that understands the characteristics of the client computer and the nearby network. It can adapt incoming data flows to customize them for the computer's particular capabilities or to match the characteristics of the data flow to the limitations of the network.

One of the most advanced proxy solutions is the Berkeley proxy [Fox97]. This system uses cluster computing technology to provide a shared proxy service for a wide variety of PDAs used at UC Berkeley. The proxy is capable of providing many important services, including transformation (changing the data from one format to another), aggregation (combining several pieces of data into one), caching, and customization (typically converting a data format into one suitable for a particular PDA). The Berkeley researchers have investigated methods of composing adaptations on a single machine [Gribble99]. They have also examined how to use a clustered proxy service to provide highly reliable, scaleable services to a large number of customers.

The Berkeley proxy design, like all other proxy systems, assumes a single point of adaptation, at the proxy server. Mechanically, multiple proxy sites can work on a single data flow, but the proxy paradigm provides no assistance in making them cooperate.

Proxy solutions vary in their degree of transparency. Most require the user to designate a chosen proxy site. Some also require special coding or alteration of programs, though others work with unaltered code.

2.2 Transformer Tunnels and Protocol Boosters

In many cases, the most effective way to handle difficult network conditions is to alter the behavior of the communication protocol. For example, if a wireless network charges money for each packet sent, consolidating small packets into larger packets before sending them over that network would be desirable. As another example, transmissions over a noisy link may benefit from adding redundant error correcting codes to the packets sent by the standard protocol.

Transformer tunnels [Sudame98] and protocol boosters [Mallet97] are two technologies that have demonstrated the benefits of this approach. Transformer tunnels use IP tunneling to alter the behavior of a protocol over a troublesome link. Generally, the method is used to provide protocol-level adaptations, such as consolidation of packets, scheduling of transmissions to preserve battery power, encryption, lossless compression, and buffering. Transformer tunnels usually work with TCP, generally precluding adaptations that fundamentally and permanently alter the contents of a data packet. Transformer tunnels are transparent to applications, but do not provide support for composition of adaptations.

Protocol boosters are modules inserted into protocol graphs to handle difficult links. Normally, their adaptations are transparent to the underlying protocol and the user and application. If not, they are deployed in pairs, with one booster performing a reversible adaptation and the other undoing it before the packets are presented to the next node or link. One sample use of protocol boosters is to insert redundant error correction packets on the incoming end of a noisy link. A paired booster on the other end strips off the error correction packets, possibly using them to regenerate any real packets that were corrupted by the noise.

Protocol boosters are composable, but the booster system does not provide support for determining if a given set of protocol boosters will perform well together. Generally, protocol boosters are assumed to provide lossless adaptations, since the system provides no support for ensuring reliable delivery if some packets are dropped or permanently altered.

2.3 Active Networks

Active networks are an attempt to add substantial amounts of adaptivity into the network infrastructure [Tennenhouse96] [Wetherall98]. In the active network paradigm, potentially each packet would execute special code at each visited router to determine its proper handling. In some active network models, the scope of this special code is extremely limited to a set of useful options. In others, any arbitrary action is permitted within security and resource limitations imposed by the network infrastructure.

Active networks thus provide an extremely general adaptation mechanism. Key design issues remain unsolved for active networks, including security mechanisms, costs, and proper architectures. Active network researchers are only beginning to look at issues of composibility of adaptations and reliability of their adapted data streams. In the long term, active networks may offer a superior way to solve the problems of adapting data streams of all forms in all circumstances, and to deploy the kinds of facilities discussed here. However, the success of this networking paradigm is not yet certain, and usable implementations of active networks are not currently available.

2.4 Application-Aware Adaptation Methods

While transparency of adaptation has many advantages, a well-designed application prepared to deal with varying network conditions is likely to perform better in important circumstances. Several groups have produced key system services for designing and building applications that participate in adapting to changing conditions.

The Rover toolkit [Joseph95] assists in designing applications to work in a mobile environment, focusing particularly on issues of varying and limited connectivity. Rover employs two key concepts, queued RPC and relocatable dynamic objects, to support mobile computers' network operations. Queued RPC allows RPC requests to be delayed until connectivity permits their completion. Relocatable dynamic objects allow a service to migrate between client and server to interact with a service, avoiding trips across a weak link. These facilities allow recoded programs to achieve substantial improvements in key performance metrics. Rover is designed primarily to deal with communications between a single mobile client and a fixed server across one bad link. It contains no explicit support for composing different adaptations (though its programming model would certainly allow composition), and it requires reprogramming for applications to use its tools.

Odyssey [Noble97] is a system service designed to support applications on mobile computers that expect to deal with varying network services. Odyssey pays particular attention to the issues of supporting multiple networking applications on a single mobile computer simultaneously, and to the value of cooperation between the applications and the operating system. Applications register their needs with Odyssey and provide upcalls to invoke when their needs can no longer be met. Odyssey wardens mediate between applications and servers, performing caching, for example. Wardens understand specific details of particular types of data flows and adaptations. The Odyssey viceroy controls resource sharing among multiple adaptations on a single mobile machine. When conditions change (either better or worse), the viceroy invokes the upcalls registered by the applications, informing them of the current limitations on resource usage. These upcall notifications allow applications to adapt their behavior to match the current conditions.

Odyssey has demonstrated significant benefits to applications sharing the same device and network, again showing that application participation in adaptation can provide valuable improvements. Odyssey is intended to deal with one difficult link between the client and server. Individual applications and wardens can compose adaptations in ad hoc ways, but support is not provided for composition of adaptations occurring at other nodes. Reliability can be very high at the destination node, since applications can be coded to deal with various kinds of failures, but failures elsewhere are not addressed.

2.5 Comparison to Related Work

The systems discussed above have demonstrated the value of adapting data flows for varying networks, and have shown the practicality of the concept for realistic situations. Conductor builds upon them, providing a framework for dynamic deployment and management of distributed adaptation.

Unlike proxy solutions, Conductor allows adaptations to occur at multiple locations in the network. This capability is shown (Section 4.4) to provide major advantages in realistic circumstances.

Unlike transformer tunnels and protocol boosters, Conductor allows lossy adaptations and provides assistance in composing adaptations. Also, Conductor has an end-to-end reliability model that these methods lack.

Conductor is based on existing, widely deployed network technology, unlike active networks. Deploying Conductor on a node requires a single, small kernel modification, rather than a complete change in the underlying networking paradigm.

Conductor does not require applications to be re-coded or even recompiled. While it forgoes some of the possibilities that systems like Rover and Odyssey exploited, Conductor has the advantage of working with off-the-shelf applications.

Despite these differences, it should be stressed that Conductor was built with the lessons learned from all of these systems in mind. Many aspects of Conductor leverage this earlier work.

3 Conductor Design and Implementation

Conductor was designed to support many styles of data communications. For instance, mobile computers may choose to communicate with arbitrary partners, some of which may also be mobile. In many cases, the networks used to transmit the data may exhibit a wide set of problems at any point. In particular, the partners, the networks, the links used, and the problems encountered may be difficult to predict. In the absence of direction from the user or the application, the Conductor system will strive to deliver the data at the highest possible quality. Since many types of data flows (e.g., HTTP, video streams, e-mail) are largely self-identifying, Conductor should be able to determine what sort of data is being transported and choose adaptations suitable for the data type automatically.

3.1 Conductor Design Principles

- Application unawareness – Conductor should assume that, in general, applications are unaware of the characteristics of the networks they use and the problems they encounter. Conductor thus does not rely on any assistance from the application. Conductor does not even expect the application to flag particular data transmissions as suitable subjects for adaptation.
- Arbitrary adaptation – Conductor should support any form of adaptation that proves useful. In particular, Conductor's design characteristics should not rule out classes of adaptations, such as lossy compression or pre-fetching.
- Distributed adaptation – For important cases, adaptation must occur at multiple points in the network in support of a single data flow. In principle, Conductor should allow adaptation at every node or router visited by the data flow. In practice, Conductor must

be prepared to work with the subset of nodes or routers that are willing and able to participate.

- Composability of adaptations – In general, a data flow may benefit from multiple adaptations applied at different locations in the network. Conductor must handle both mechanical and semantic implications of passing the data flow through these multiple adaptors.
- Planning – Conductor's ability to select a set of adaptors that improve the user's experience is key to its success. Conductor must create a plan for the deployment of adaptors in support of a data flow. Since this plan may include composition of several adaptations, Conductor must ensure compatibility between adaptations.
- Reliability – Many useful forms of adaptation alter the data, sometimes even removing some content. Protocols like TCP expect every bit sent to be delivered, eventually. Conductor must ensure that semantically meaningful data is reliably delivered, despite adaptations that may alter the data's form.
- Easy deployability – Conductor should be easily integrated with a popular, widely used system, and should require minimal alterations to that system.

3.2 Conductor Architecture

Conductor is a stream-oriented adaptation service intended to be present on various nodes in a network. Preferably these nodes will be at or near gateways between networks of differing characteristics, so adaptation modules can be deployed at these points. Conductor consists of two main pieces: adaptors, and the framework for deploying those adaptors.

Conductor adaptors are self-contained pieces of code that perform some particular adaptation, often only for a particular type of data stream. The set of Conductor adaptors is expandable. Each Conductor node might have a different set of adaptors available for local use. Adaptors are frequently (although not necessarily) paired, converting from a given protocol to a protocol better suited to the transmission medium, and back to the given protocol.

By conforming, at the endpoints, to the protocol expected by the user application, Conductor is able to provide an application transparent service. However, paired adaptors need not regenerate the original data flow, nor are they necessarily user-transparent. Adaptors may deliver any data to the application, so long as it conforms to the expected protocol. For instance, an adaptor may cause a color image to be transformed to a black-and-white image, or frames to be dropped from a video stream. These adaptations will clearly affect the user's experience.

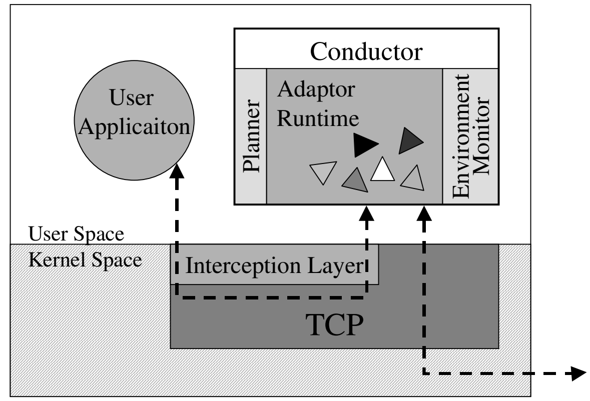

Figure 1: The Conductor architecture deployed on a node.

Conductor provides a framework to support the operation of adaptors. Figure 1 shows the architecture of Conductor on a single node, consisting primarily of a user-space module that handles monitoring of data flows, delivery of data streams to local adaptors, transmission of data streams between Conductor nodes, planning for new data flows, and recovery and reliability. In addition, in most systems Conductor requires a small kernel modification to trap new data flows, allowing Conductor to examine them for possible adaptation and initiate planning. In some systems, existing extensibility mechanisms may allow trapping of data flows without kernel modifications [Mosberger96].

When a new data flow is started by an application (which is unaware of the presence of Conductor and of the prevailing network conditions), Conductor traps the opening of its socket. Conductor currently only traps TCP sockets, but can be extended to handle other protocols. Conductor examines information about the socket (and possibly information about the first few bytes of data sent to the socket) to determine if the system understands the format of the data well enough to handle it. Conductor has moderately heavy setup costs, so it will usually not try to assist extremely short data streams. Assuming Conductor does understand the data format and expects that the setup costs will be dominated by the adaptation benefits, it effectively kidnaps the TCP socket, providing the illusion of end-to-end TCP, when actually Conductor is handling the reliable end-to-end delivery of data.

Once Conductor has chosen to intercept a connection, it must form a path over which data will flow. Presumably this path will contain both Conductor-enabled and non-enabled nodes. Conductor follows the normal routing path and probes for Conductor-enabled nodes along the way. As this path of potential adaptation sites is formed, information about local network conditions and node capabilities are gathered from each Conductor node discovered and forwarded along the path. Once the path is formed, therefore, the information required to generate a plan has been collected at the destination node. This information is used to generate a plan for which adaptors to deploy. This plan is then delivered back to the participating nodes in one round-trip message, causing a data path to be created with the appropriate adaptors inserted.

Conductor requires some method of reliably delivering bits from node to node. Currently, Conductor uses TCP for communication between Conductor nodes. Effectively, Conductor splits the end-to-end TCP connection into individual high-level node-to-node TCP connections, while providing the required end-to-end services itself. In the future, Conductor could also make use of other protocols specifically designed for particular link characteristics, such as WTCP [Sinha99].

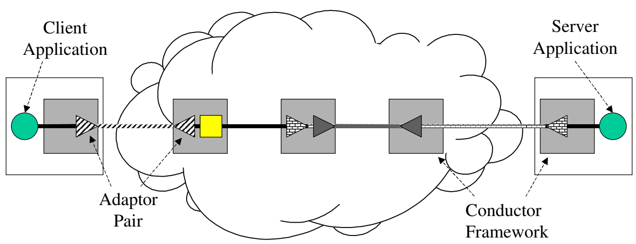

Figure 2: Conductor intercepts client-server communication channels and deploys distributed adptors.

Once the path is set up, Conductor forwards the user's data stream down the path. Figure 2 gives a simple view of Conductor in use. At each Conductor node, an adaptation might be applied to the data. Some adaptations do not change the data, but many do. Potentially, the bits that arrive at the destination may be very different than the bits that were sent. However, if Conductor's planner has done its job properly, the arriving bits are the most suitable, semantically meaningful version of the data that was possible to deliver in the face of prevailing network conditions. In the video example, dropping color in the face of limited bandwidth yields black-and-white frames that are semantically related to the color image that was sent, but the overall sets of bits are very different.

Conductor monitors the data path during the course of the data flow. In the current implementation, Conductor is primarily interested in extreme variations in the available resources and failures. Adapting to minor variations of bandwidth, delay, etc., is the job of the individual adaptors. If, however, the variations are too large for them to handle, or if there is an actual failure, Conductor will signal a problem. The effect of this signal is to initiate replanning. Conductor may try to find a new data path, or alter the set of deployed adaptors on the old path.

3.3 Conductor Planning

Deciding which adaptors to deploy on behalf of an application, in what order in the communication sequence, and on what machines, can be easy or extraordinarily difficult. If there are only a modest number of applicable adaptors and potential execution sites, and if interaction between adaptors can be ignored, a planning algorithm can be easy to construct. In contrast, designing an algorithm in a complex, multi-hop environment can be difficult due to order dependent adaptor decisions, computation limits at certain sites, and a wide family of deployable adaptors. If the planning algorithm takes too long, circumstances may change, causing the plan to be outdated before it is even completed.

Conductor does not make the planning problem any worse; the problem exists for each application in a complex networked environment. Nor does Conductor solve the planning problem. However, a framework like Conductor does provide an environment for deploying good heuristics and evaluating planning methods. Also, Conductor provides an efficient mechanism to gather the information required by planning, and mechanisms for implementing the chosen plan.

Much research has been done in the general area of planning to solve complex problems with varying constraints [Lever94] [Veloso98]. This paper does not intend to make a contribution to this research, or even to use the most sophisticated planning algorithms already developed. Conductor currently uses a simple planning algorithm that works well for many important situations. Future research will examine more sophisticated planning for network adaptivity.

Currently, Conductor uses a centralized planning procedure. Conductor collects a description of conditions and problems that are present on the nodes and the links that will host a particular data flow. The description might also contain user suggestions about the kinds of adaptation the user might prefer, or other constraints on Conductor's behavior. The planning module applies rules and heuristics to this data to build a plan.

When Conductor decides to serve a new data flow, it must select a set of Conductor nodes, forming a path between the application client and server. Information must be collected from each Conductor node along the path. Conductor gathers local information from the node initiating the communication and sends it to the next node in the path. That node adds its own information, and forwards the collection to the next node. The information collected at each node includes relevant local link conditions (such as link bandwidth, delay, and jitter), node capabilities (processing speed and storage size), and a list of available adaptors. Eventually, the information reaches the destination node. The destination node is the first node that has all relevant information available, so Conductor performs planning here, running an algorithm on the collected information. The resulting plan should generally be better than planning performed incrementally, with partial knowledge, at each Conductor node.

Each adaptor has a static record, containing the information the planner needs about the adaptor's behavior. This record includes the format the adaptor accepts and produces and other properties, such as whether and how the adaptor alters the compressibility of the data. This and similar properties allow the planner to avoid the error of trying to compress encrypted data, or applying Lempel-Ziv compression to an image before attempting to drop color information. The adaptor properties also describe the resources that will be consumed during its execution.

The use of an adaptor description imposes an overhead on the adaptor writer, who must prepare the record. Reasonable care is required, or poor decisions will result from even the best of planning algorithms.

3.3.1 Conductor Planning Algorithm

The Conductor planning algorithm consists of two major steps. First, Conductor associates link problems with candidate adaptors that handle these problems. For example, if we need to send more bits on a particular link than its bandwidth actually permits, some form of compression should be performed. Matching problems to adaptors is done on a per-link basis. Simultaneously, the planner verifies the ability of the nodes to run the selected adaptors.

Selecting the proper adaptor to handle a problem depends on general observations, user preferences, and planning criteria. For example, assume that a user wants to send a 1Mbps real-time video data stream and the only channel available is a modem running at 56Kbps. Conductor needs to select adaptors that can reduce the amount of data 16 times, while minimizing the amount of data lost. Conductor might first find the best matching lossless compressor, perhaps reducing the data by 50%. Since further data reduction is required, Conductor might then choose the best matching color-dropping adaptation, reducing the amount of data by another 75%. As a last resort, Conductor might choose a frame-dropping adaptation that will drop every other frame, achieving the required 16 times reduction. Conductor would also determine the proper order of applying these three adaptations. Conductor follows the user's guidelines to constrain its selections. For instance, the user may prefer to drop resolution rather than color.

The choice of adaptors at this stage of the planning algorithm may affect the later stages of the algorithm. Poorly chosen adaptors might prevent reaching the optimal plan because of latency and resource mismatches. The current Conductor planner uses a static set of rules to match problems and adaptor solutions.

The second step of Conductor planning resolves the problems of composability and resource matching for the global end-to-end plan. Optimizing the initial plan requires merging similar adaptors and extending the scope of adaptors where appropriate. The algorithm for this step follows:

1. Create the ordered set of adaptors *A{}*, initially empty.
2. For each link, scan all adaptors selected for this link**.** For the next adaptor a_i scanned:
   - 2.1. Verify the composability of a_i and all rightmost adaptors from *A{}*.
   - 2.2. Verify that the resources required by a_i match those available at the node.
   - 2.3. Check if a_i can be merged with adaptors in *A{}* or extended over more links.
   - 2.4. If any constraint is violated, return ERROR, otherwise add a_i to *A{}*.

At the end of the algorithm, *A* will contain the set of adaptations tied to the particular nodes where they should be executed. Note that this algorithm can result in an ERROR return. Currently, in such cases, Conductor will not deploy any adaptors. A more sophisticated planning algorithm would include some form of backtracking to find another plan if the initial attempt fails.

3.4 Conductor Reliability

Conductor decomposes the single, end-to-end TCP link into multiple TCP links between the adaptations, since the data content is being altered, otherwise confusing TCP. However, Conductor's use of split TCP breaks the end-to-end reliability semantics normally provided by TCP. Without further support, failure of a Conductor node would cause the failure of all connections passing through that node. Moreover, Conductor interposes potentially stateful adaptor modules into the data stream. For example, many compression algorithms retain information about data already processed to assist in processing subsequent data. Failure of any one of these modules could also result in connection failure. To protect against these types of failures, Conductor provides an additional end-to-end reliability model.

Adaptation, however, complicates end-to-end reliability by removing the assumption that data is immutable in transit. Typical reliability mechanisms attempt to provide exactly-once and in-order delivery of each byte transmitted. Since adaptor modules can arbitrarily change the data stream as it passes through each Conductor node, the bytes received can differ arbitrarily, in number and kind, from the bytes transmitted. Attempting to provide exactly-once delivery is futile.

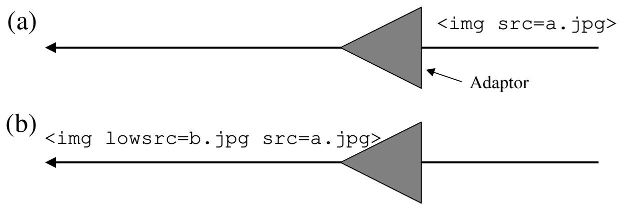

Figure 3: Adaptation of an HTML tag – (a) initial data enters adaptor, (b) adaptor adds *lowsrc* tag.

For instance, consider the stream of bytes in Figure 3a representing a tag from an HTML document. An adaptor module might choose to insert a new attribute to this tag, as in Figure 3b. If the adaptor subsequently fails, perhaps before the entire tag is delivered to the destination, we would need to determine a point of retransmission. Using a byte-count, for example, would lead to reception of neither the original HTML tag nor the adapted version (see Figure 4). Since the state in the adaptor, describing the change, has been lost, it is no longer possible to determine an appropriate point of retransmission.

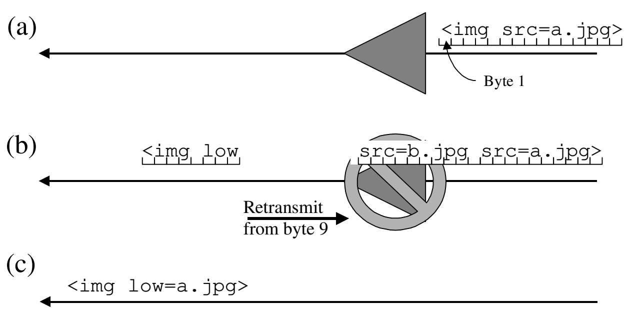

Figure 4: Failure recovery using a byte-count – (a) data arrives at adaptor, (b) failure and retransmission occur, (c) retransmission produces an undesirable result.

3.4.1 *Semantic Segmentation*

In Conductor, we have chosen a new model of reliability that is compatible with adaptation: exactly-once and in-order delivery of semantic meaning. In the above example, it is clear that either the original tag or the adapted tag provide the same semantic function in the overall HTML document. Adaptation has merely altered the form of that semantic meaning. *Semantic segmentation* allows an adaptor to provide enough information to ensure that each semantic element in the data stream is delivered exactly-once and in-order, even if the adaptor was to fail.

As the name implies, semantic segmentation breaks the data stream into segments. A segment is the basic re-

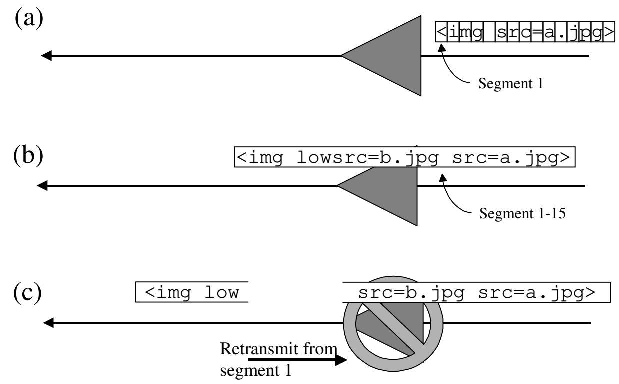

Figure 5: Failure recovery using Semantic Segmentation – (a) one byte segments arrive at adaptor, (b) segment combination and adaptation occurs, (c) failure requires segment recovery.

transmission unit of Conductor. Initially, the data stream can be thought of as being logically segmented into one-byte segments, as shown in Figure 5a. Adaptors wishing to modify the data stream must contain their changes within individual segments. If a change would cross two or more segments, those segments must first be combined into a single segment. The new segment must maintain all of the semantic meaning of the segments it replaces. In our previous example, before adding a new attribute to the tag, the adaptor would first combine the segments making up the tag into a single segment, as shown in Figure 5b.

Note that the framing overhead for semantic segmentation can be extremely low. Only a few bytes are required to track each segment. Moreover, segments can be of arbitrary size. Also, although the initial stream is logically considered to be a sequence of one-byte segments, such streams can actually be transmitted as the original byte-stream, without requiring per-byte framing.

3.4.2 *Failure Recovery*

When a node, link, or adaptor fails, it is only necessary to determine which segments have been completely received downstream of the failure. Segments that are partially received are discarded. Retransmission begins with the segment following the last complete segment. Note that retransmission may also imply readaptation, which does not necessarily produce the same byte-stream as before. Any form of the original byte-stream which emanated from the source, adapted or not, can replace segments that were lost. In the above example, the partially received segment is thrown away and retransmission begins with segment 1 (see Figure 5c), thus preserving the semantics of the tag.

Retransmission is triggered by a retransmission request which follows the data path in reverse to the source, passing through all Conductor nodes and adaptors. Conductor nodes and adaptors can allocate data caches and satisfy retransmission requests from these caches, or they can forward the request. Since applications are unaware of Conductor and cannot respond to retransmission requests, a cache of the original data stream generated by the application must be provided at the data source. Once retransmission begins, the data can be adapted as before, or in any manner now appropriate.

Since application servers are not aware of segmentation, data from a partially received segment can not be delivered to the application. Only when the segment is complete can it be delivered. Then, an acknowledgement is sent back toward the source. The acknowledgement is cumulative and indicates that this segment and all previous segments have been received at the endpoint. This acknowledgement allows adaptors and nodes to free any cache space or other state relevant to acknowledged segments.

3.4.3 Preserving Proper Composition

Adaptors are frequently interdependent. Failure of an adaptor generally requires that it be replaced or that the hierarchy of adaptors be altered. Since arbitrary adaptation algorithms are allowed, a given adaptor may maintain state. Therefore, it may not always be possible to simply reinstantiate an adaptor. For instance, replacing a compression adaptor may require it to build a new dictionary, which is no longer compatible with the downstream decompression adaptor. Even when appropriate, reinstantiation of an adaptor is not always possible. If a node fails, the system may be unable to locate the code or another node to run it.

When it is not possible or appropriate to reinstantiate an adaptor, the paired adaptor must be removed. In addition, any adaptation composed in the failed adaptation will no longer receive the input it expects and must also be removed. Finally, any caches on nodes between the paired adaptors must be invalidated, since they too will contain data in a now unknown format.

3.5 Implementation Details

Conductor was developed on top of Linux 2.0. The framework is primarily written in Java. Adaptors are also written in Java.

Conductor intercepts TCP streams generated by local applications through the use of a loadable kernel module that allows a new set of functions to be stacked on top of the normal socket functions for the TCP protocol. It was necessary to add one function to the Linux kernel to support the stacking of socket interfaces. The new socket functions allow Conductor to modify the parameters of the application's `connect()` call, causing it to connect to the local Conductor framework instead of the remote server. The interception layer also allows Conductor to determine the destination originally requested by the application and to maintain the illusion that the application client is actually connected to the application server.

Conductor makes use of the transparent proxy facility, present in Linux as part of the kernel's firewall feature, to discover the Conductor nodes between a client and server. A client sends a UDP packet to the server. The first Conductor node along the normal route to the server intercepts the packet and then forwards it along to discover the next node along the path.

Adaptors make use of an API that provides them with access to the data stream and limited inter-adaptor communication capabilities. An adaptor uses an AdaptorWindow object to operate on a chunk of the data stream. The AdaptorWindow object provides various flavors of two main operations: `expand()` and `contract()`. The `expand()` operation allows the adaptor to add more bytes to the upstream end of the window from the data flow. The `contract()` operation allows the adaptor to push bytes downstream, out of the window, and on to another adaptor. An adaptor can operate on the data stream using one or more DataAccessPointer objects, which provide byte-related access and modification operations while maintaining the rules of segmentation. Finally, adaptors can pass data to other adaptors of the same stream, or other streams, via an inter-adaptor communication cache.

4 Conductor Performance

To evaluate Conductor performance in actual deployment, we present a sample application where Conductor might be useful. We then report the performance results of the experiments designed to mimic that application. All of the results are presented with a 90% confidence interval.

4.1 Performance Test Environment

The real-life scenario we chose and considered more closely is mobile Internet accesses (Figure 6).

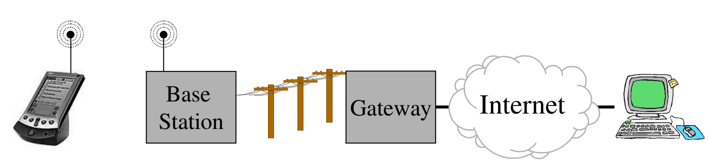

Figure 6: Test environment. A hand-held device communicates to a server through a wireless link, a modem, and the Internet.

An untethered hand-held device communicates to a base station, which in turn connects to an ISP through an expensive modem link. The ISP then forwards the traffic to the desired destination. We assert that this network topology is a plausible and representative model for a mobile Internet access infrastructure.

For the user of this hand-held device, many concerns immediately arise. The hand-held device has limited battery power. The wireless link is error prone. The modem link can be slow and costly, and small data transfers can be expensive if the charges are based on the number of connections. The Internet is known to be insecure. The list goes on.

In our experiments, each network node is represented by a Dell Inspiron 3500, with Pentium II 333 Mhz processors and 64 Mbytes of memory each. In a real system, the various components would have widely differing capabilities. In particular, the hand-held device would have limited CPU computing resources and limited battery life. These assumptions were fed to the planning algorithm. For the wireless link, we used 2 Mbps AT&T WaveLAN cards with a power consumption specification of 3.00, 1.48, and 0.18 Watt seconds for corresponding transmitting, receiving, and sleeping modes [Rudenko98]. We used a 56 Kbps PPP serial connection to emulate the modem link and 10 Mbps Ethernet between the ISP and the destination Internet server. Although representing the Internet and server with dedicated hardware is not realistic, the results will tend to understate the benefits of Conductor.

4.2 Description of Application

The actual application using this network environment allows users to perform image database queries. In our particular example, archaeologists wish to use this application in the field to submit visual queries to the distant image database server, which will return 24-bit color images that match the submitted queries. We developed this system based on the needs expressed by archeology researchers [Ancona97]. The application was written with no knowledge of underlying network infrastructures, and it uses the standard TCP socket API to exchange visual queries and image results. The application was specifically written for this experiment, and many features that would be required in a real system that are irrelevant to the experiment were not implemented.

Each visual query of an archaeological artifact consists of a contour sketch, a color sample, and a texture sample image of the artifact. The image database returns up to three images that best match the query. The query and results exchange in a rendezvous fashion: each query blocks until the matching images return.

4.3 Experimental Settings

We randomly chose 20 queries as the benchmark load. Queries and results have average sizes of 310 (± 52) and 700 (± 81) Kbytes, respectively. The same 20 queries were repeated across different adaptor deployment settings.

Our experiments assume that end-to-end response time, throughput, and power consumption are the primary concerns. The end-to-end response time is defined as the time between the beginning of a query transmission from the hand-held device to the end of receiving corresponding results. Response time does not consider the planner costs because those costs are not paid at every query (Section 4.5). Throughput is the number of completed queries over time. If we consider queries as being submitted back-to-back, the throughput will trend as the inverse of response time. Power consumption is the power required by the network interface device for transmitting, waiting, and receiving a query at the hand-held device.

We directly measured response time. Power consumption was calculated based on measured times and the rated power consumption specification of the WaveLAN cards. Experiences with measuring power consumption have shown that the reported life of a battery is highly unreliable and subject to non-linear variations [Rudenko98], so this method provides a more accurate picture of power consumption than would querying the battery.

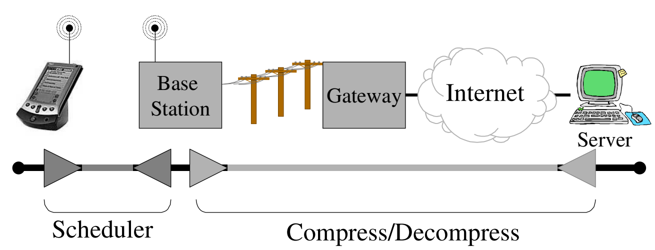

Figure 7: Application network environment with adaptors deployed.

We would not expect this application to perform particularly well in this environment without assistance. The modem link will significantly slow image transmission, and much battery power will be wasted waiting for results that will be arriving very slowly. Conductor can improve the performance of the application by deploying a set of adaptors (see Figure 7).

Compression and scheduler adaptors are two candidates for improving the performance of this application. Compression adaptors can improve the end-to-end response time by reducing the size of image representations. However, the time to perform the compression has to be less than the time saved in transmission. Since our queries have rendezvous semantics (as opposed to non-blocking semantics), compression might also improve throughput by reducing the per-query response time. Reducing the transmission time also decreases the power required for queries. The compression adaptors in our experiments use Lempel-Ziv compression.

The scheduler adaptor is responsible for turning on and off the network device on the mobile computer to save power while waiting for the arriving data stream. However, turning off the network device longer than the waiting period might adversely affect both response time and throughput. The implemented scheduler currently approximates the waiting interval based on a variant of a moving average over recent waiting times.

4.4 Conductor in Action

We ran experiments in several configurations, including the cases without Conductor, with Conductor with no adaptors, and with Conductor running with either or both scheduler and compression adaptors in place. Because of limited computing resources on the handheld computer, the planner chooses to place the compression and decompression adaptors between the base station and the image database server, in both directions. In order to reduce the communication lag perceived during scheduler coordination, the planner chooses to place the scheduler between the hand-held device and the base. The planner described in Section 3.3 was actually running and making decisions on adaptor deployment in these experiments.

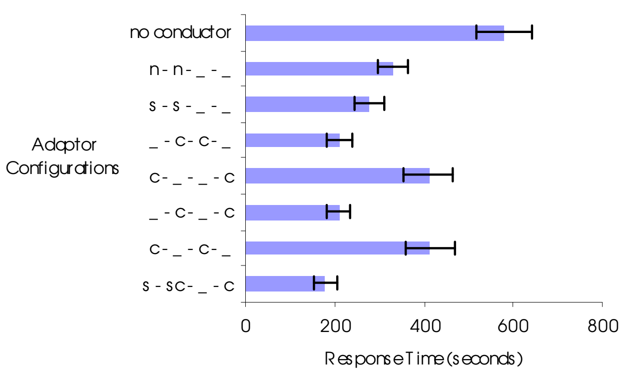

Figure 8: Conductor response time for various adaptor configurations.

Figure 8 shows end-to-end response times for various adaptor configurations, with "no Conductor" as the base comparison. In the "no Conductor" case, simple TCP communications were used.

In these diagrams and the following text, we use special notation to indicate which adaptors were deployed for each case. As shown in Figure 6, the data passes over three links: a wireless LAN, a dialup line, and the Internet (considered as a single link for simplicity). Adaptors could be deployed at any link endpoint. Our notation is shorthand indicating which adaptors are deployed at each location. In this notation, each link is represented by "-". Characters between the link symbols indicate which adaptors are deployed. "_" means no adaptors are deployed. "n" means a null adaptor is deployed. Null adaptors receive all incoming data by Conductor, but pass it back again, unaltered and as quickly as possible. "c" indicates a compression or decompression adaptor. "s" indicates a scheduling adaptor. Scheduling adaptors must be deployed on both sides of a link to be effective.

Deploying both scheduler and compressor adaptations (s - sc - _ - c) reduces response time by 69%. Intuition suggests that the 55% image compression ratio achieved by Lempel-Zev compression of the data in our benchmark workload should account for the majority of this improvement. However, our measurements on the effects of each adaptation showed the contrary.

Either compression adaptation (_ - c -_ -c) or scheduling adaptation (s - s _ - _) reduced response time, by 63% and 52% respectively. One might expect the scheduler to occasionally overestimate the arrival times of results and subsequently increase response time. To better explain the scheduler behavior, we replaced the scheduling adaptors with null adaptors (n - n - _ - _), still resulting in a measured response time improvement of 43%. The majority of response time reduction by the scheduler adaptation is obtained because of the split in the TCP connection at the base-station (note that split-TCP is only used when adaptors are deployed on a node). Packets lost due to errors in the wireless link can be retransmitted more quickly from the base station than from the server because they do not have to travel over the high-latency PPP link. This effect has been previously reported in [Cohen98].

The remaining improvements achieved by the scheduling adaptors are primarily due to the bulk transmission of data. When a packet is lost, another packet follows quickly behind, producing a duplicate acknowledgement and triggering fast retransmission of the lost packet. We checked this hypothesis by replacing the wireless link with an Ethernet link. Without the wireless link, the scheduling adaptation had no significant affect on response time, as expected.

Conventional wisdom would suggest that extending compression (primarily intended for the modem link) across more of the network would improve performance, since other links would also need to handle less data. We would thus expect the end-to-end compression (c - _ - _ - c) and the compression over the problematic links (c - _- c - _)* to perform better than the other two configurations (_ - c - c - _ and _ - c - _ - c). However, the latter configurations performed better. Running one end of the

* This case corresponds to a typical positioning of adaptors when using a single proxy.

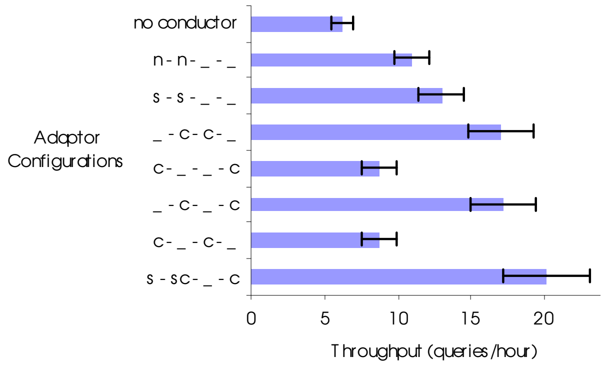

Figure 9: Conductor throughput for various adaptor configurations.

compression adaptor at the base station has the side effect of causing Conductor to run split TCP across the wireless LAN link. Even without the scheduling adaptor, running split TCP across this link allows quicker retransmission of lost packets.

The throughput portrait, shown in Figure 9, demonstrates benefits similar to response time. The full adaptor deployment case (s - sc - _ - c) generates 3.2x improvement in throughput.

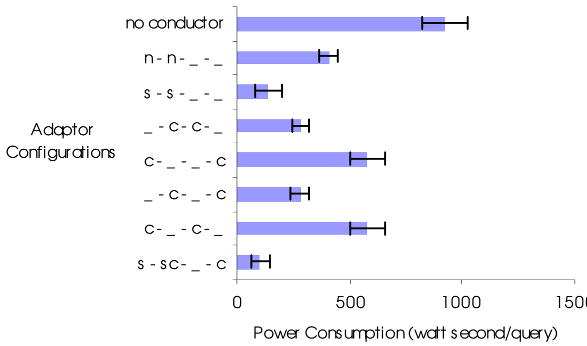

Figure 10: Conductor power consumption for various adaptor configurations.

With respect to power consumption (Figure 10), deploying our adaptors (s - sc - _ - c), saves power by a factor of 10.

The split TCP adaptation at the base station (n - n - _ - _) improves power saving by a factor of 2.5. Since both the scheduler (s - s _ - _) and compression (_ - c - _ - c) adaptors implicitly carry the benefits of splitting TCP, we need to evaluate their effects relative to the null adaptation case to isolate their contributions. Relative to the null adaptation case, the scheduler and compression adaptations improve power savings by factors of 2.8 and 1.4, respectively. By multiplying the three isolated factors, we roughly obtain the power saving factor of 10 for the combined case (s - sc - _ - c).

Power savings by various compression adaptor configurations are direct results of reducing the response time. Since current scheduler adaptors predict the result arrival time at approximately 70% accuracy, we could achieve greater power savings by improving the scheduler accuracy.

4.5 Conductor Overhead

The most important overheads exhibited in Conductor will tend to vary depending on exact circumstances, such as the kinds of links in the network and the power and load of the nodes running Conductor. Thus, generalizing about these overheads is difficult. In the sample application scenario, the current Conductor implementation exhibits 30% increase in latency, 25% reduction in throughput, and 16% extra power consumption, without any adaptation. If Conductor decides the stream is not conducive to adaptation at all, the overhead would be significantly lower. As shown above, when Conductor chose to perform these experimental adaptations, their introduction more than overcame these overheads.

Detailed measurements indicate that using native threads, rather than user-level threads, will remove over 90% of the above overheads. Conductor currently uses user-level threads because the Java native thread library for our Java environment contains serious bugs. We plan to switch to the Java native thread library once these bugs are removed, leading to substantial performance improvement.

Planning is performed once per connection, not necessarily for every query and response. In the case of the experiments reported here, there were 20 queries per connection. Therefore, the cost of planning is amortized over multiple queries. The cost of planning in these experimental runs was typically 139 milliseconds or approximately 7 milliseconds per query. The average improvement in response time was about 399,350 milliseconds, easily dominating the planning cost.

It is also encouraging that the cost of adding null adaptors is quite small, less than 0.1% impact on throughput and latency. The measured impact of the (necessary) use of split TCP is also modest, generally under 2%.

4.6 Performance Comments

Our experience with Conductor indicates a substantial opportunity to improve network service by strategic deployments of adaptations. At the same time, the effects of interactions of different adaptations for an application in a specific network environment are reasonably complex. For example, even we were surprised by the impact of split-TCP effects on our experiments. This experience adds weight to the view that application writers cannot realistically be expected to effectively take into account the significant impacts of underlying details and assumptions in a complex network.

The results of our experiments also suggest variations on conventional wisdom. Multiple point adaptation appears superior to single proxies in accommodating the variety of needs of complex modern networks. Simple, end-to-end adaptations have less flexibility to tailor solutions to fit the heterogeneous link characteristics of multi-hop network paths. Overall, composition of adaptors introduces new ways to solve a variety of network problems simultaneously within a single framework.

Of course, these views must be tested in a wide set of networking circumstances and applications before one should conclude that applications would benefit enough to justify wide deployment of a framework like Conductor. Nevertheless, the observed improvements are substantial enough, given an unoptimized code base, that the approach appears quite promising.

5 Applicability of Distributed Adaptation

There are numerous approaches to adaptation in computer networks, as suggested in Section 2. However, accelerated growth in the scale and heterogeneity of networks, provide considerable incentive for a general solution that both covers a wide range of possibilities and can be extended to address emerging network challenges in the future.

Early networks were relatively homogeneous. However, today one can anticipate a truly ubiquitous network presence, leading to increased network heterogeneity. Inexpensive home LANs, metropolitan-area wireless access, personal-area wireless devices, public access networks, smart buildings, high bandwidth multimedia paths, and even ad hoc dynamically deployed networks will all be interconnected. Personal portable devices (e.g., PDA, phone, watch) will communicate and coordinate in local activities, as will devices in a car or office (e.g., laptop, desktop, printer, A/V equipment, a visitor's PDA). These clusters of network devices will desire connectivity with other clusters of communication via perhaps a nearby public access point, the wired workstation, or a wireless device such as the cell-phone or a packet radio.

For instance, a heart monitor may use the cell-phone to contact a doctor's PDA to report an anomaly, along with relevant data, while the doctor is at lunch. The PDA might be connected via the restaurant's public wireless LAN that is in turn connected to the network via the city's metropolitan-area network. This connection might cross numerous networks with widely varying characteristics.

This increased network complexity and diversity require that more systems-level self-management be present in the network. Some information required for proper adaptation is unlikely to be available at the endpoints; instead it will be present only at gateways within the network. For instance, only the cell-phone knows if it currently has an established connection (and is therefore virtually free to use).

In such an environment, one cannot expect applications to shoulder the burden of adaptation. The more complexity in future networks, the more relevant this observation becomes. Moreover, retrofitting legacy applications for each network evolution is impractical (and at any stage of evolution, most applications are legacy). Therefore, the network adaptation layer should be distinct from, and generally independent of, the application layer in the network stack. The need to largely decouple solutions to network complexity from application design is analogous to abstracting the complexities of hardware via device drivers, virtual memory, and other operating systems services.

Application-specific adaptation benefits greatly from being deployed into the network. Some adaptations, such as forward error correction, caching, and the scheduler used as an example in Section 4, must be placed at particular nodes in the network to be effective. The placement of other, more general adaptations such as compression, may seem best at endpoints, but their location may be restricted by load balancing concerns, security restrictions, or resource constraints [Vahdat99].

It should be noted, however, that a general solution to transparent distributed adaptation requires a corresponding reliability solution. The use of non-trivial adaptors in the network changes the content of a data-stream and thus breaks the reliable delivery guarantee of a protocol like TCP, upon which so many applications depend. One must reconstruct that guarantee to remain transparent. Fortunately, it can be cheaply provided by adding a modest amount of mechanism and reusing underlying TCP services.

6 Future Work

There are numerous areas that require solutions in order for distributed adaptation to be widely employed. Several important items are discussed below.

6.1 Security

Conductor needs a security layer, not only to protect the user's data through encryption (which requires a suitable key distribution protocol), but also as a means to protect against unwanted adaptations. That is, it is important to make the framework itself safe and robust in the face of security challenges. Also, distributed adaptation normally requires that a stream be decrypted before adaptors can operate on the data. Either one must trust the framework to do so or else choose the order of adaptations carefully so that the data is available in clear text when needed. We are currently developing a security extension that allows Conductor nodes to agree upon and execute a variety of different models for authentication and key distribution, based on the level of security required by the user.

6.2 Planning

The problem of planning is a rich research topic in itself. Conductor's existing planning algorithm is far from perfect. There are many possible improvements, including:

- Multicasting node conditions and planning preferences for greater efficiency
- More involvement by local nodes in the planning process
- Reuse of cached plans for commonly occurring conditions
- Incremental planning to provide quick approximate plans that are improved as more time becomes available to examine different options
- Evaluating the merits of alternate transmission paths

The existing Conductor planning method mostly serves to demonstrate that the planning framework is adequate to make and implement automated decisions.

6.3 User and Application Control

Conductor permits user-input into the planning process to help determine which adaptors to select based on which data characteristics are most important to the user at this moment. Conductor's current user control interface is extremely limited, and human factors are important in this case. A richer interface is not necessarily better. Instead, the interface must present options in human terms: the impact of dropping B-frames in a video transmission is probably little known to the average user. An effective interface for user control over adaptation is an open issue.

At the same time, although Conductor can transparently make many decisions on behalf of applications, knowledgeable programs could clearly give Conductor better advice than its planner could deduce on its own. In addition, such "Conductor-aware" applications could provide a more seamless interface for users to register their preferences regarding an application's behavior.

We plan to improve both aspects of Conductor.

6.4 Performance Analysis

The current performance of Conductor and the existing measurements are encouraging, but more work is certainly necessary. The implementation needs to be tuned to demonstrate, in practice, that distributed adaptation can impose little overhead. Furthermore, measurement in a wide set of circumstances is needed to provide high confidence that the specifics of the viewpoints expressed in this paper bear up to generality.

7 Conclusions

New technologies increase the complexity of computer networks and make network behavior difficult to predict. A trend toward network diversity only increases the complexity, and as a result, applications can no longer expect a consistent level of service from the network. At the same time, users want applications that work properly and without undue cost in the network environments of the future. Users will not accept applications that fail to take network diversity into account.

Even if it were feasible, periodically updating an applications ability to react to unfavorable network conditions would not be effective. To adapt effectively, a presence within the network is required to monitor the link characteristics and to adapt the data stream on the user's behalf.

A single point of adaptation in the network is also insufficient in complex networks. We have demonstrated, by example, that placing adaptation at multiple locations across the network can provide a considerable improvement in complex networks that cannot be achieved using a single proxy. This example represents a wider class of scenarios in which user data must cross several links with differing characteristics.

Support for distributed adaptation requires an appropriate framework for monitoring the characteristics of interior network nodes and links, determining and deploying a set of compatible adaptations, and preserving the expected reliability semantics. Existing solutions do not prove all of these characteristics. In particular, none allow the coordination of multiple adaptations among multiple nodes. Extending any of these approaches to support distributed adaptation would require similar functionality and overheads to Conductor.

Conductor demonstrates the feasibility of building a system to support distributed adaptation. Conductor provides protocol-level adaptation to tailor an application's use of a network, and thus the user's experience, in an application-transparent manner. Beyond the simple mechanics of properly deploying multiple adaptation modules, Conductor provides two key capabilities to support distributed transparent adaptation.

- Conductor includes a planning infrastructure that allows an end-to-end picture of the network to be gathered, possible solutions to be evaluated, and a set of adaptors to be deployed. We have demonstrated the use of this infrastructure with one possible planning algorithm that selects appropriate adaptations in a few important cases.
- Conductor introduces a new model of semantic reliability that allows arbitrary adaptation while preserving the expected end-to-end reliability semantics.

We have demonstrated the benefits of using Conductor with a sample application based on real-world requirements. This example shares characteristics with a wide class of applications, suggesting that Conductor can provide benefit to many important applications. Without Conductor, use of our example application in the field could be costly, severely limiting the benefit to the user. By deploying adaptations into the network in a distributed manner, Conductor is able to significantly reduce both the response time and power requirements of the application, allowing the user to obtain faster results and work longer on a single battery charge.

References

[Ancona97] M. Ancona, G. Dodero, C. Fierro, V. Gianuzzi, V. Tine, A. Traverso, "Mobile Computing for Real Time Support in Archaeological Excavations," *Proceedings of Computer Applications in Archaeology*, University of Birmingham, UK, April 1997.

[Balakrishnan95] H. Balakrishnan, S. Seshan, E. Amir, and R. Katz, "Improving TCP/IP Performance Over Wireless Networks," *Proceedings of the 1st ACM International Conference on Mobile Computing and Networking (MobiCom '95)*, November, 1995.

[Cohen98] R. Cohen and S. Ramanathan, "Using Proxies to Enhance TCP Performance over Hybrid Fiber Coaxial Networks," Hewlett-Packard Laboratories Tech Report #HPL-97-81, 1997. Available at: http://www.hpl.ph.com/techreports/97/HPL-97-81.html

[Fox97] A. Fox, S. Gribble, Y. Chawathe, E. Brewer, and P. Gauthier, "Cluster-Based Scaleable Network Services," *Proceedings of the 16th ACM Symposium on Operating System Principles*, October, 1997.

[Gribble99] S. D. Gribble, M. Welsh, E. A. Brewer, and D. Culler, "The MultiSpace: an Evolutionary Platform for Infrastructural Services," to appear in *Proceedings of the 1999 Usenix Annual Technical Conference*, Monterey, CA, June 1999.

[Joseph95] A. Joseph, A. deLespinasse, J. Tauber, D. Gifford, and F. Kaashoek, "Rover: A Toolkit for Mobile Information Access," *Proceedings of the 15th ACM Symposium on Operating System Principles*, December 1995.

[Lever94] J. Lever and B. Richards, "parcPlan: A Planning Architecture with Parallel Actions, Resources, and Constraints," *Proceedings of the, 8th International Symposium on Methodologies for Intelligent Systems (ISMIS '94)*, Berlin, Germany, Springer-Verlag, 1994, pp. 213-222.

[Mallet97] A. Mallet, J. Chung, and J. Smith, "Operating System Support for Protocol Boosters," HIPPARCH Workshop, June 1997.

[Mosberger96] D. Mosberger and L. Peterson, "Making Paths Explicit in the Scout Operating System," *Proceedings of OSDI '96*, October 1996, pp 153-168.

[Noble97] B. Noble, M. Satyanarayanan, D. Narayanan, J. Tilton, J. Flinn, and K. Walker, "Agile Application-Aware Adaptation for Mobility," *Proceedings of the 16th ACM Symposium on Operating System Principles*, October, 1997.

[Rudenko98] A. Rudenko, P. Reiher, G. Popek, and G. Kuenning, "Saving Portable Computer Battery Power through Remote Process Execution," *ACM Mobile Computing and Communication Review (MC2R),* Vol. 2, No. 1, 1998.

[Sinha99]P. Sinha, N. Venkitaraman, R. Sivakumar, and V. Bharghavan, "WTCP: A Reliable Transport Protocol for Wireless Wide-Area networks," to appear in the *Proceedings of the Fifth Annual International Conference on Mobile* Computing *and Networks (MobiCom '99)*, August 1999.

[Sudame98] P. Sudame and B. Badrinath, "Transformer Tunnels: A Framework for Providing Route-Specific Adaptations," *Proceedings of the Usenix Technical Conference*, June 1998.

[Tennenhouse96] D. Tennenhouse and D. Wetherall, "Towards an Active Network Architecture," *Computer Communications Review*, April 1996.

[Wetherall98] D. Wetherall, J. Guttag, and D. Tennenhouse, "ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols," IEEE OPENARCH'98, April 1998.

[Vahdat99] A. Vahdat, M. Dahlin, T. Anderson, A. Aggarwal, "Active Names: Flexible Location and Transport of Wide-Area Resources," to appear in *Proceedings of the Second* Usenix *Symposium on Internet Technologies and Systems*, Boulder, CO, October 1999.

[Veloso98] M. M. Veloso, M. E. Pollac, and M. T. Cox, "Rationale-based Monitoring for Planning in Dynamic Environments," *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*, Menlo Park, CA, AAAI Press, 1998, pp. 171-179.