Automatic Creation and Reconfiguration of Network-Aware Service Access Paths

Xiaodong Fu and Vijay Karamcheti
Department of Computer Science
Courant Institute of Mathematical Sciences
New York University
{*xiaodong,vijayk*}*@cs.nyu.edu*

Abstract

A promising approach for providing seamless service access to portable and mobile end-devices is to augment the network path between client applications and services using "bridging" components that are capable of caching, protocol conversion, transcoding, etc. While several such path-based approaches have been proposed, current approaches lack mechanisms for (1) automatically creating effective network paths whose performance is optimized for encountered network conditions, and (2) dynamically reconfiguring such paths when these conditions change. This paper describes our work on addressing these shortcomings. Our approach, which is built into an application-level programmable network infrastructure called CANS (Composable Adaptive Network Services), relies on modeling enhancements and algorithms to construct augmented network paths that not only improve application performance by coping with the resource gap between network services and clients, but can also dynamically adapt to changes in the network environment.

We evaluate our approach over a range of network and end-device characteristics using two application scenarios: web access and image streaming. Our results validate the effectiveness of our approach for enabling network-aware service access to mobile clients, verifying that (1) data paths automatically created with our path creation algorithm do bring applications with considerable performance advantages; (2) fine tuned, desirable adaptation can be achieved using our flexible component model and reconfiguration strategy; and that (3) despite their flexibility, the run-time overhead of generated data paths is negligible, and the cost of path reconfiguration is small enough for most applications to continually adapt to dynamic changes.

Key words: Adaptive middleware, Middleware for ubiquitous and mobile computing, Communication adaptation, Programmable networks.

1 Introduction

Advances in wireless networking and communication-enabled portable devices such as lightweight laptop computers, PDAs, and cell phones, raise the prospect of a mobile user being able to interact with network-based services in a seamless, ubiquitous fashion. To consider a scenario, a mobile user who initiates a teleconference using a laptop at his office desk can continue to participate in it even when he needs to step away from his desk or altogether leave the building, relying upon a wireless LAN in the first case and a metro-area or cellular wireless network in the second.

However, several challenges need to be addressed before this vision can become reality. First, many services assume that they will be accessed by relatively powerful clients using high bandwidth, low latency connections. This assumption is at odds with the low-bandwidth networks and resource-constrained portable devices used by mobile clients. Furthermore, a mobile user may experience very different connection characteristics over time, arising from user mobility or dynamic cross-traffic in shared network environments. Ideally, the user's interactions with the service should continually adapt to such changes.

Unfortunately, current solutions in widespread use that rely either on differentiated service for different user groups, or a close coupling between the service and client applications to adapt to changing network conditions, are incapable of ensuring this. Differentiated services, used in popular news and stock trading services, cannot satisfactorily handle users with connections exhibiting big variations in available bandwidth (which may be caused by increased load when a large amount of new users join in the shared network or when users move away from the access point). On the other hand, the approach of encoding adaptation logic into client and server applications, exemplified by automatic stream selection mechanisms in most commercial media players, requires considerable programming effort and detailed knowledge of network state, and usually do not cope well with dynamic network changes. Moreover, adaptation using such end-point mechanisms usually come with high overhead, especially in cases where long network paths are used. Finally, the fact that adaptation logic is hard coded into the client/server makes it hard for such approaches to be extended to cope with new problems.

More promising are several recent projects [4, 5, 7, 14, 16, 21–23, 25], which have proposed augmenting the network path between client applications running on diverse end-devices and network services with application-specific components that can be dynamically inserted and modified as required by the encountered network conditions. The components in such *path-based approaches* can transparently handle stream degradation, reconnection, and in general support arbitrary transcoding, caching, and protocol conversion operations, serving to "impedance match" an end-device with the network service. Although there have been a large number of proposals, most path-based systems have focused primarily on providing system support to allow dynamic insertion and deletion of components, with little attention paid to determining, without user involvement, which components should be present along the path and how they should be mapped to the intermediate nodes. The few projects that have addressed this latter issue have focused only on coping with the heterogeneous nature of the last-hop link and end device. Thus, current path-based approaches suffer from two limitations that limits their wider applicability. First, the approaches lack an effective mechanism for *optimizing* data communication performance to the encountered network conditions (which requires more than just satisfaction of heterogeneity constraints). Second, no current approach addresses the challenge of continuously *adapting* the path, as network conditions change. Addressing this challenge requires both coping with data in transit along a prior path as well as the run-time overheads of path reconfiguration.

This paper describes our work on these two problems. Our approach, which is built within a programmable network infrastructure called CANS (Composable Adaptive Network Services), focuses on *application-neutral* mechanisms for automatic path creation and reconfiguration that provides applications with augmented network paths whose performance is optimized for the underlying network conditions and which can dynamically adapt as these conditions change. These mechanisms rely on both an enhanced modeling of component behaviors than encountered in current path-based approaches, as well as algorithms that take advantage of these models.

To evaluate our approach, we have conducted a series of experiments with a Java-based CANS prototype implementation, using two representative applications: web access and image streaming in environments with different network and end-device characteristics. The results validate our approach, verifying that (1) automatic path creation and reconfiguration are achievable and do in fact yield substantial performance benefits; (2) our approach is effective for providing applications that have different performance preferences with fine tuned, desirable adaptation behaviors; and that (3) despite the flexibility, the overhead incurred in augmented data paths is negligible; the cost to reconfigure data paths is acceptable for most applications, and can be further reduced by employing local version of our mechanisms.

Specifically, the contributions of this paper include:

- A model for abstracting component behaviors and an algorithm using this model to construct data paths with optimized performance for general path based approaches. The creation of such paths requires only a high level specification of the application's performance preferences and underlying network conditions.
- System support for *low-overhead dynamic path reconfiguration*, providing applications with semantic continuity on data transmissions. Moreover, the paper describes a local reconfiguration mechanism, permitting each portion of the network path to adapt independently and concurrently to network changes while maintaining overall performance requirements for the whole path.
- Evaluation of the path creation and reconfiguration strategies using representative applications. This evaluation has been critical in helping us first identify and then fix shortcomings with our approach.

The rest of this paper is organized as follows. Section 2 briefly reviews related approaches that have investigated augmentation of network paths and introduces the overall CANS architecture and the underlying system assumptions. Sections 3 and 4 describe in turn the modeling enhancements and algorithms for automatic path creation and reconfiguration. Section 5 evaluates these mechanisms using the two applications. Section 6 summarizes the differences between our automatic path creation approach and that adopted by other path-based systems, identifies avenues for future work, and discusses limitations of CANS-like approaches. We conclude in Section 7.

2 Background

2.1 Related Work

Our work is related to a large body of previous work that has proposed augmentation of the network path between client applications and network services with “bridging” components.

Odyssey [17], Rover [10], and InfoPyramid [15] are examples of systems that support end point adaptation. Each system provides only minimal support for composing adaptation activities across multiple nodes, and consequently may not be flexible enough to cope with weak devices or changes in intermediate links.

Systems such as transformer tunnels [21], protocol boosters [14] are examples of application-transparent adaptation efforts that work at the network level. Such systems can cope with localized changes in network conditions but cannot adapt to behaviors that differ widely from the norm. Moreover, their transparency hinders composability of multiple adaptations. More general are programmable network infrastructures, such as COMET [4], which supports flow-based adaptation, and Active Networks [22,23], which permit special code to be executed for each packet at each visited network element. While these approaches provide an extremely general adaptation mechanism, significant change to existing infrastructure is required for their deployment.

General path based approaches such as Active Proxies [5], CANS [7], Ninja [8], Scout [16], and Conductor [19] are more flexible in that they can dynamically insert, map, and modify application-specific components along the network path between services and client applications, while building on existing network-level infrastructures. However, as stated earlier, the state-of-the-art of such approaches includes only limited mechanisms for optimizing performance of constructed paths for different network conditions, and does not at all address the reconfiguration of paths as network conditions change.

Our objective in this work is to address these limitations, which we believe hamper the wider applicability of path-based approaches. We have additionally required that these mechanisms be *application-neutral* in an effort to achieve maximum flexibility without overly burdening the application developer. Our approach is built within the CANS infrastructure, but should be applicable to all general path based approaches including the systems mentioned above. In the rest of this section, we briefly review the CANS architecture and make explicit our system assumptions.

2.2 Overview of the CANS Architecture

The **C**omposable **A**daptive **N**etwork **S**ervices (CANS) infrastructure [7] views network environments as consisting of client *applications* and *services*, connected by *data paths*. CANS extends the notion of a data path, traditionally limited to data transmission between end points, to include dynamically injected application-specific components, called *drivers*. Serving as the basic building block for CANS paths, drivers are standalone *mobile* code modules that can be connected via a standard *data port* interface.

As shown in Figure 1, the CANS network is realized by partitioning service and driver components belonging to data paths onto physical hosts, connected using existing communication mechanisms. Data processing code in a driver is executed in CANS Execution Environments (EE) that run on hosts along the network route.

To ensure that components are connected together in a valid fashion, CANS relies on a notion of type compatibility similar to some other path-based approaches [8]. Data types used in CANS are augmented with additional attribute information and custom compatibility operators. E.g. a MPEG type, with a frame size attribute, is defined as being compatible with MPEG types with the same or smaller frame size. Each CANS component is modeled as an entity that transforms the types on its input ports into types on its output ports. Compatibility constraints arise from components requiring that their input types satisfy certain requirements: two components can be composed if and only if the type produced by one is compatible to that required by the other.

2.3 System Assumptions

The CANS infrastructure was designed with a shared wireless network environment in mind, of the kind encountered in an office, airport, hotel, or shopping mall setting. Thus, CANS currently makes some assumptions that hold for such closed environments.

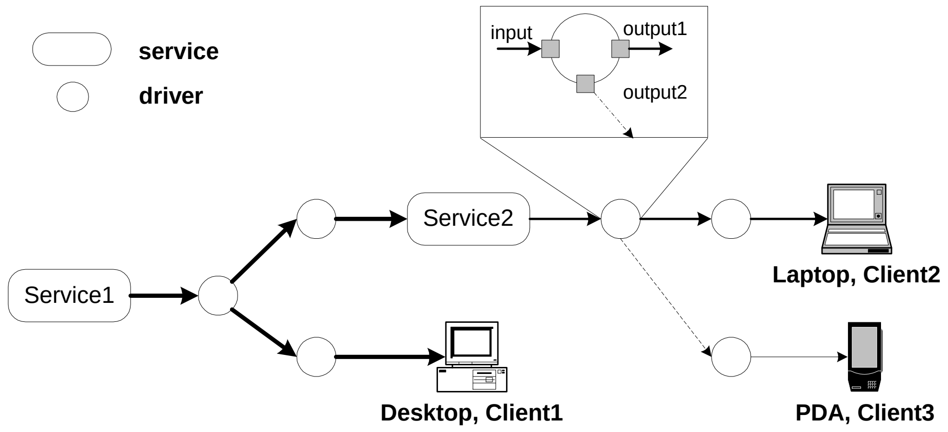


Figure 1: Logical view of a CANS network showing data paths constructed from typed components.

- First, we assume that the hosts underlying the CANS network either belong to the same network administration domain or that there is an implicit trust relationship between the various domains, hence security concerns are not explicitly addressed in the current CANS implementation.
- Second, meta-information (type information) about possible driver components is assumed available in local directory services, and to involve a closed set of semantically unambiguous types.
- Finally, we rely upon the presence of modules external to the CANS infrastructure to dynamically monitor resource availability, to allocate available resources among existing augmented paths (and enforce this allocation), and to inform a path about changes in its resource allocation as appropriate. While we realize that current-day, best-effort networks do not satisfy these assumptions, we believe that they are a prerequisite to supporting guaranteed or differentiated service. In this work, our focus has been on how a path should be constructed or be adapted given a particular resource allocation for it. We note that other research projects are looking into estimation of network resource availability over different timescales [11,13] and that there are industry-led standardization efforts such as IEEE 802.11e for bandwidth allocation and enforcement in wireless networks [18,20].

We will revisit these assumptions in Section 6.

3 Automatic Path Creation Strategy

In general, creation of an augmented data path consists of two steps: *route selection* where a graph of nodes and links is selected for deploying the path, and *component selection and mapping* where appropriate components are selected and mapped to the selected route. Route selection is typically driven by external factors (such as connectivity considerations of the wireless hop, ISP-level agreements, etc.) and so we focus only on the component selection problem here.

At a high level, the component selection and mapping problem requires that given a specification of application preferences and network conditions, we should be able to *automatically* identify which components ought to make up the augmented path and where they should be mapped. Given our objective of optimizing the path performance in accordance with the underlying network conditions, we need ways of characterizing the impact of a particular component on the resource utilization along a path as well as for associating a performance metric with the overall path. Additionally, driven by efficiency considerations, we would like to support creation and reconfiguration of sub-portions of the path at a time, without requiring complete knowledge of the whole path.

Unfortunately, it is not simple to satisfy these requirements if one were to restrict oneself to just the type compatibility notions of the base CANS infrastructure (and other state-of-the-art path-based approaches). While type compatibility can capture heterogeneity constraints on component deployment (e.g., that a component cannot be deployed on a node unless it satisfies certain restrictions, or that a certain component is required to bridge a certain kind of link), coming up with an combined selection and mapping strategy that optimizes performance requires both enhancements to the base component model as well as algorithms that leverage this model. We describe these two aspects of our path creation strategy below.

3.1 Modeling Enhancements

The base CANS component model needs to be extended in several ways to support both automatic selection of components according to the various network constraints as well as to characterize the resource utilization and performance of a path. We describe these extensions in additional detail below:

Stream types The notion of data type compatibility in the base CANS model is unable to capture the situation where a component that produces a generic type is required to connect to a component that requires a more specific type. To see how such situations can arise, consider the behavior of a path that links up a service to a client application using a compression and a decompression component pair to overcome network bandwidth limitations. The component pair can operate on arbitrary data types at the input but must produce a specific type on the output to permit valid connections. The information about this specific type is available if one looks at the entire path, but then this precludes independent adaptation of sub-portions of the path.

Our approach copes with this problem by defining the notion of a *stream* type, which captures the aggregate effect of multiple CANS drivers operating upon a data stream, and is represented as a *stack* of data types. Components are modeled as transforming the stream type at their input into stream types at their output. In our example above, the stream type at each point in the path would contain information about the original (specific) type available at the input to the compression-decompression pair. Keeping this information explicitly in the type stack enables each part of the data path to adapt independently.

Data type ranks To express application-specific constraints on how components can be composed together (e.g., that for data paths requiring both encryption and compression that encryption happens after compression to ensure better bandwidth reductions), we rank each data type and require that only types of monotonically increasing ranks can be stacked into a stream type. To capture the above constraint, it suffices to give the encryption data type a higher rank than the compressed data type. Note that data type ranks also reduce the size of the search space that must be explored in the component selection and mapping algorithm described below.

Modeling of network resources Although the notion of type compatibility defines the space of possible CANS paths (each type-compatible sequence that transforms the service-provided type into a type required by the client application), sometimes certain components may need to be present on valid data paths to satisfy constraints imposed by network environments. For example, encryption/decryption components are needed for data transmission across insecure wireless links if privacy is required by the application. Our approach expresses such constraints by introducing the notion of *augmented types*, which extends data types with environment properties. Network resources are modeled as entities that transform the environment properties of augmented types in a type-specific fashion. For example, an insecure link changes the “privacy” property of a data type, while an encryption component is capable of insulating the “privacy” property of types from being affected by the environment. Modeling both application data type and resource constraints using a unified framework has the advantage that valid paths are simply captured using the notion of type compatibility on the augmented types. Our automatic path creation strategy exploits this fact.

Component resource utilization model To characterize the resource utilization and performance of a path, we need to capture the behavior of each component without requiring an explicit enumeration of all possible situations in which the component can be mapped. To facilitate this, each **driver** d is modeled in terms of its *computation load factor* ($\mathrm{load}(d)$), the average per-input byte cost of running the component, and its *bandwidth impact factor* ($\mathrm{bwf}(d)$), the average ratio between input and output data volume. For example, a compression component that reduces stream bandwidth by a factor of two has a $\mathrm{bwf} = 0.5$ and the corresponding decompressor has $\mathrm{bwf} = 2.0$.

This simple model can be extended to allow components to have multiple configurations. Further, for each configuration, the values of computation cost and compression ratio are determined by the actual stream type of incoming data. For example, when an image resizing driver is placed after an image filtering driver, its load and bwf factors are determined by the image quality attributes contained in the type object generated by the filtering component. Such values can be obtained by an approach we call *class profiling*, which basically groups possible value of these data properties (for our example, the image quality) into several classes, and profiles components with representative data in each class. Values between different classes are estimated using linear interpolation.

3.2 Component Selection and Mapping Algorithm

Our path creation strategy leverages the above modeling enhancements to automatically select and map a type-compatible component sequence to underlying network resources. In addition to satisfying type requirements, the strategy respects constraints imposed by node and link capacities and optimize network path performance according to application requirements. In general, selecting a set of components and mapping them to a network route to obtain strictly optimal performance metrics (such as highest throughput or shortest latency) is a NP-hard problem. Fortunately, with a few reasonable simplification assumptions, the problem becomes more tractable.

We restrict our attention to single input, single output components; i.e., all selected plans consist of a sequence of components. Most of the application scenarios we have experimented with fall into this category. The heart of our strategy is a dynamic programming algorithm. We first describe a base version of the algorithm in which a single performance metric, in particular, data throughput, needs to be optimized. After that we discuss its extension to handling *value ranges* (i.e., that the path performance metric lie within a range of values), which is the typical performance preference for most streaming applications. Finally, we describe the local planning mechanism, which allows disjoint segments of a data path to be configured independently and concurrently while maintaining the overall performance guarantee for the whole path.

3.2.1 Base Algorithm

To describe the dynamic programming algorithm, we first need to introduce some terminology.

A **data path**, $D = \{d_1, \ldots, d_n\}$, is a sequence of type-compatible components. Type compatibility is defined in a **type graph** G_t: a vertex ($t \in V(G_t)$) in the graph represents a type, and an edge $e = (t_1, t_2) \in E(G_t)$ represents a driver that can transform data from type t_1 to type t_2.

A **route**, $R = \{n_1, n_2, \ldots, n_p\}$, is a sequence of nodes. Each node n_i is modeled in terms of its computation capacity, and a link between two adjacent nodes is modeled in terms of its bandwidth capacity. Both capacities are defined in terms of route resources available for a particular path.

A **mapping**, $M : D \rightarrow R$, associates components on data path D with nodes in route R. We are only interested in mappings that satisfy the following restriction: $M(d_i) = n_u, M(d_{i+1}) = n_q \Rightarrow u \leq q$, i.e., components are mapped to nodes in path sequence order.

The component selection process takes as its input a route R, a type graph G_t, a source data type t_s, a destination data type t_d, and attempts to find a data path D that transforms t_s to t_d and can be mapped to R to yield maximum throughput.

As we mentioned before, the problem as stated above is NP-hard. To make the problem tractable, we take the view that the computation capacity can be partitioned into a fixed number of *discrete* load intervals; i.e., capacity is allocated to components only at interval granularity. This practical assumption allows us to define, for a route R, the notion of an *available computation resource vector*, $\vec{A}(R) = (r_1, r_2, \ldots, r_p)$, where r_i reflects the available capacity intervals on node n_i (normalized to the interval [0,1]).

In the description that follows, we use p for the number of hosts in route R (i.e. $p = |R|$); m for the total number of types (i.e. $m = |V(G_t)|$); and n for the total number of components. It is safe to assume that $m < n$.

Dynamic Programming Strategy
The intuition behind the algorithm is to incrementally construct, for different amounts of route resources, optimal solutions with $i+1$ (or fewer) components, using as input optimal partial solutions involving i (or fewer) components. Because of our mapping definition, if the $(i+1)^{\text{th}}$ component d_{i+1} is assigned to a node n_k in the route, drivers before d_{i+1} can only make use of resources on the nodes n_j with $j \leq k$. Consequently, only resource vectors of the form $(1, ..., 1, r_j \in [0,1], 0, ..., 0)$ need to be considered in this step. These set of resource vectors is designated RA.

Formally, the algorithm fills up a table of partial optimal solutions ($s[t_s, t, \vec{A}, i]$) in the order $i = 0, 1, 2, \ldots$. The solution $s[t_s, t, \vec{A}, i]$ is the data path that yields maximum throughput for transforming the source type t_s to type t, using i components or fewer and requiring no more resources than $\vec{A}(\vec{A} \in \mathrm{RA})$. The algorithm is shown in Figure 2. Line 3 of the algorithm handles the base case: only the case $t = t_s$ achieves non-zero throughput. Lines 8–13 represent the induction step, examining different drivers to extend the current partial solution for each specific intermediate type t and resource vector $\vec{A}$. Lines 12 and 13 ensure that the driver achieving the maximum throughput defines the next-level partial solution.

```
Algorithm Plan
Input: $t_s$,$t_d$, $G_t$, $R$
Output: The data path that yields maximal throughput from type $t_s$ to $t_d$ on route $R$
1.  (* Step 1: Initialization for partial plans with zero components *)
2.  for all $t, \vec{A} \in \mathrm{RA}$
3.      do calculate $s[t_s, t, \vec{A}, 0]$
4.  (* Step 2: Incrementally building partial solutions *)
5.  for $i \leftarrow 1$ to $p \times n$
6.      do for all $t \in V(G_t), \vec{A} \in \mathrm{RA}$
7.              do $s[t_s, t, \vec{A}, i] \leftarrow s[t_s, t, \vec{A}, i-1]$
8.                  for all $d = (t', t) \in E(G_t)$
9.                      do for all $n_j$ that $\vec{A}[n_j] > 0$
10.                             do $M(d) \leftarrow n_j$
11.                                $\vec{A}' \leftarrow (\vec{A}[0], \ldots, \vec{A}[n_j - 1], \vec{A}[n_j] - \mathrm{load}(d), 0, \ldots)$
12.                                if $\mathrm{throughput}(\mathrm{append}(s[t_s, t', \vec{A}', i-1], d, \vec{A})) > s[t_s, t, \vec{A}, i]$
13.                                  then $s[t_s, t, \vec{A}, i] \leftarrow \mathrm{throughput}(\mathrm{append}(s[t_s, t', \vec{A}', i-1], d, \vec{A}))$
14. return $s[t_s, t_d, \vec{A} = [1, 1, ..., 1], p \times n]$
```

Figure 2: Base Path Creation Algorithm

The throughput for a particular mapping can be computed given the node throughput and link bandwidth properties. Node n_i's throughput itself is decided by the incoming bandwidth, its computation capacity $\mathrm{comp}(n_i)$, and the load and bwf properties of components mapped to the node.

The algorithm terminates at Step $p \times n$. This follows from the observation that there is no performance benefit from mapping multiple copies of the same component to a node. The complexity of this algorithm is $0(n^2 \times m \times p^3) = 0(n^3 \times p^3)$ as opposed to $O(p^n)$ for an exhaustive enumeration strategy. As stated earlier, in most mobile access scenarios, p is expected to be a small constant, with overall complexity determined by the number of components.

Two implementation issues need additional attention here. First, reducing the size of type graph is important. When calculating paths, only types that can be reached from both source and destination types are considered. In addition, type ranks (described in Section 3.1) can be used to further reduce the size of type graph. These mechanisms help because of the observation that the total number of possible composable operations involving a specific type is limited. Second, when a type object needs to be made available across a network link, the augmented part of the type object needs to be calculated on the other side of the link as described in Section 3.1.

3.2.2 Extension 1: Planning for Value Ranges.

Instead of seeking the minimal/maximal value on a single performance metric, many applications require the value of a performance metric to be in an *acceptable range*. Only after that range has been met does the application worry about other preferences. For example, most media streaming applications usually demand a suitable data transmission rate (in some range) so that media data can be rendered appropriately at display devices; once the transmission rate is kept in that range, other factors such as data quality become the concern for the application. We use the terms *range metrics* and *performance metrics* to refer to the two types of preferences.

Given that our planning algorithm constructs data paths by incrementally filling in a solution table of $s[t_s, t, \vec{A}, i]$, it is natural to extend this to check that retained solutions satisfy two conditions: (1) values of range metrics achieved on the current solution will lie within the desired range, and (2) the value of any performance metrics is in fact optimized.

Although this is the basic idea of the extension, for some range metrics, such as path latency, additional work is needed. For such range metrics, even if the current value of the range metrics is not in the range for a partial solution, this does not exclude the possibility that this partial path may actually become a part of the final solution. For example, appending compression components to the partial path can bring down overall path latency by reducing packet size. So such candidates cannot be pruned. To *estimate* whether the desired range can in fact achieved by appending additional components, we employ a procedure called *complementary planning*, which just runs the planning algorithm in reverse, providing information about whether or not the range metrics can reach the desired range using residual resources along a data path that transforms type t to t_d. Note that complementary planning needs to be run just once. Heuristic functions are used for choosing among candidate paths that can all meet the required range.

3.2.3 Extension 2: Local Planning for Segments of the Network Route.

For CANS-like mechanisms to be used in large scale networks, the key is to allow local decisions, which can not only enable agile adaptation to changes in the network, but also make such systems be easily used/deployed in the situation where the path spans multiple network domains(we will revisit this point in section 6). This necessitates a distributed mechanism in which each individual node or a partial segment of the path can independently and concurrently make *local* decisions on how to adapt. To support this, a selection and mapping algorithm for a segment of an existing network path is needed. The challenge here is doing so while still being able to maintain some overall performance guarantee: for example, that the range metrics for the entire path will still fall within their desired range. Note that local mechanisms, though they enhance responsiveness of data paths, may compromise optimality of performance metrics, but we look at this as a reasonable tradeoff.

Our local planning strategy is a straightforward extension of the range planning mechanism described earlier. To create a partial path for R', which is a segment of the original route R, all we need to do is to run the range planning algorithm on R' with localized parameters. Since the type before and after R' is fixed by the type entering and leaving R', the only thing left is to adjust the range metrics for R'. Adjustment for throughput and latency is shown below:

- For applications that require an overall throughput range $[\mathrm{th}_{\mathrm{low}}, \mathrm{th}_{\mathrm{high}}]$, the adjusted throughput range for R' stays the same but we assume a throughput of $\mathrm{th}_{\mathrm{high}}$ at the input point of R'. The intuition is that planning with $\mathrm{th}_{\mathrm{high}}$ will ensure that R' remains capable of achieving the desired throughput range for any input throughput in the range of $[\mathrm{th}_{\mathrm{low}}, \mathrm{th}_{\mathrm{high}}]$.
- For applications that require a latency range $[l_{\mathrm{low}}, l_{\mathrm{high}}]$, the localized latency range will be $[l_{(\mathrm{low},\mathrm{R}')}, l_{(\mathrm{high},\mathrm{R}')}]$, where $l_{(\mathrm{V},\mathrm{R}')}$ is the divided portion of latency V over segment R'.

4 System Support for Efficient Path Reconfiguration

As we mentioned before, data paths may need to be reconfigured to cope with dynamic changes in available resources. We assume as stated earlier that an external module informs the path about appropriate changes in resource availability over the timescale of interest.

The main problem that needs to be resolved by the reconfiguration mechanism is how to deal with any internal state that might be present in components making up the affected segment. The presence of such state raises several challenges. First, since each component can change the content of the data packets passing through it, traditional transport-layer mechanisms which focus only on providing an in-order byte stream are not adequate. Instead, the requirement is for continuity at the semantic level. For example, if a user is browsing a set of web pages, the reconfiguration mechanism should ideally guarantee that the browser application receives complete web pages, maybe in a different format (for example, a page can be distilled to overcome low bandwidth links), but in the same sequence, with each page appearing exactly once. Second, to satisfy our objective of independent and concurrent reconfiguration in different portions of the network path, the reconfiguration mechanism must achieve this semantic continuity without relying upon complete knowledge of the whole path because doing so is likely to incur considerable overhead and increase reconfiguration time. Ideally, we would like to achieve as small a reconfiguration cost as possible so as to permit agile adaptation to network resource changes.

Our path reconfiguration protocol provides the semantic continuity guarantee for data transmission when data paths are modified. It does not required complete knowledge of the whole path, instead, only information about the reconfigured part is needed. The local reconfiguration mechanisms, which leverage the local planning mechanism described earlier, can further bring down the reconfiguration overhead. However, as with the path creation strategy, the reconfiguration strategy also requires enhancements to the component model, which we describe below.

4.1 Driver Interface Restrictions

To support dynamic path reconfiguration on portions of the overall path, we adopt the same high-level strategy that is used in transport layer solutions, namely buffering and retransmission. However, respecting application-specific data

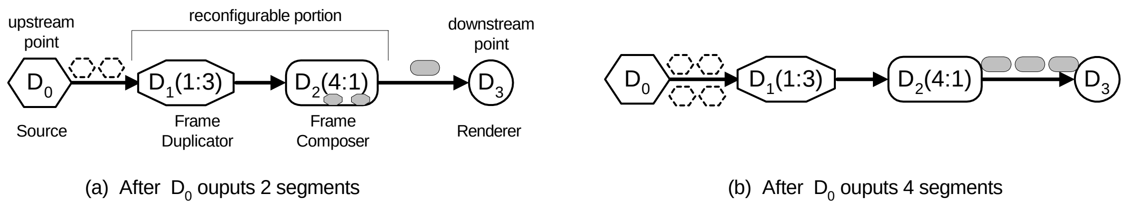


Figure 3: An example of data path reconfiguration using semantics segments.

continuity semantics requires a means for associating semantics with component stream interactions and mechanisms for recreating a stream with the appropriate semantics.

To facilitate the above, our approach places two natural restrictions on the component interface. CANS drivers are required to demarcate *semantic segments* of data they produce and satisfy a *soft state* property.

Semantic segments refer to application-specific units of data transmission, e.g., an HTML page or an MPEG frame. CANS drivers are required to (logically) consume and produce data at the granularity of an integral number of semantic segments. Note that this demarcation can be either implicit (each message unit identifying a separate semantic unit) or explicit (with marker messages identifying the boundary between two semantic units).

Soft state refers to the fact that any internal state in a driver can be reconstructed simply by restarting the driver and injecting an appropriate sequence of segments. Stated differently, given a semantically equivalent sequence of input segments, a soft-state driver always produces a semantically equivalent sequence of output segments. Components that perform transcoding, caching, or library-based compression are examples of drivers that satisfy the soft-state property.

Together, the semantic segment and soft-state properties help to

- track which output segments depend on which input segments; and
- identify the data that needs to retransmitted to ensure semantic continuity.

4.2 Reconfiguration Protocol

Reconfiguration of a portion of the data path works as follows. Upon receiving an event that triggers reconfiguration, the upstream point starts *buffering* segments while continuing to transmit them, in effect flushing out the contents of intermediate drivers.[1] The downstream point monitors the output segments arriving there, waiting until it completely receives an output segment from upstream satisfying the property that *all subsequent segments correspond only to input segments at the upstream point that are either buffered or not yet transmitted*. At this time, the system can be stopped and the data path portion replaced by a semantically equivalent set of drivers. To restart, the upstream point *retransmits* starting from the first segment whose corresponding output segment was not delivered.

Figure 3 shows an example of reconfiguration on a path, in which d_1 produces three frames upon each incoming frame while d_2 combines every four incoming frames into one. Figure 3[a] shows a situation when the path can not be reconfigured due to the internal state in d_2. The reconfiguration should be delayed to the time depicted in Figure 3[b], when the segments buffered at d_0 allow the recreation of any possible internal state in d_1 and d_2.

4.3 Local Reconfiguration

To cope with dynamic changes in networks, instead of relying on a centralized mechanism to reconfigure the path globally, it is usually much faster to allow each individual node or a partial segment of the path to independently and concurrently make *local* decisions on how to adapt. To provide such support, in Section 3.2.3 we described the algorithm for calculating new partial data paths. In this part, we will describe how to choose the reconfiguration segment (R').

The mechanism to choose a reconfiguration segment R' should consider the tradeoff between the length of the segment selected for reconfiguration (which affects reconfiguration cost), and the likelihood that the reconfiguration can handle the change (otherwise, further actions are needed where more hosts are involved). We adopt the following heuristic to make this choice. A first-level reconfiguration action happens in a local node when its load changes or

[1] *Upstream* and *downstream* points refer to the components immediately upstream and downstream of the portion being reconfigured.

the load on its direct downstream link changes. A second-level reconfiguration action includes network segments comprising nodes connected with relatively fast links. If both these fail, we fall back to global reconfiguration.

In summary, from the performance perspective, local reconfiguration can be viewed as a mechanism to balance the tradeoff between agility of adaptation and the optimality of data transmission. In network environments where change is frequent and modification of the whole data path may cause big overhead, such a mechanism can result in better user experience. From a non-performance perspective, comparing with global ones, local mechanisms can make it easier for such systems to be used in situations where multiple administrative network domains are involved. We will revisit this point in section 6 when we discuss our future work.

5 Performance Evaluation

To evaluate the effectiveness of our approach, we built the automatic path creation and reconfiguration support into the CANS infrastructure (a Java-based prototype is available for download at http://www.cs.nyu.edu/pdsg/cans), and conducted a series of experiments in the context of a web access application and an image streaming application under typical mobile usage scenarios.

First, using the web access application, we measured the performance of automatically generated paths under a wide range of network conditions.

Second, we examined the adaptation behaviors achieved with automatically generated paths, using the image application in a shared wireless environment, where available bandwidth changes frequently.

Last, we measured reconfiguration overhead of data paths, for both the global and local mechanisms.

5.1 Experimental Platform

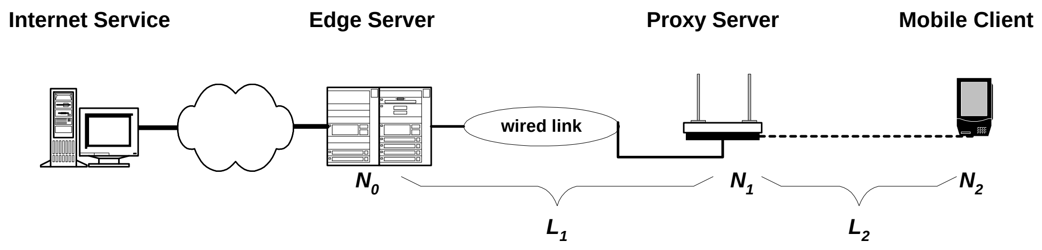


Figure 4: A typical network path between a mobile client and an internet services.

We consider a typical network path between a mobile client and an Internet server as shown in Figure 4. This platform models a mobile user using a portable device (N_2) such as a laptop or a pocket PC to access network services in a shared wireless environment. The communication path from the device to the service typically spans three hops: a wireless link (L_2) connecting the user's device to an access point, a wired link (L_1) between the wireless access point and a gateway to the general Internet, and finally a WAN link between the gateway and the host running the service. We assume that CANS components can be deployed on three sites, the mobile device (N_2), a proxy server located close to the access point (N_1), or an edge server located near the gateway (N_0).[2]

The **web access application** is a browser client, which downloads web pages (both HTML page and images) from a standard web server. For this application, short response time is preferred.

The **image streaming application** is a simple Java-based application that continuously fetches JPEG frames from an image server and display them[3]. To perform appropriately, this application requires a certain frame rate, and prefers high data quality.

[2]Our use of the term "edge server" differs from its usage in content distribution networks. We use the term to refer to a host on the frontier of the network administrative domain within which CANS components can be deployed.

[3]Although we would have ideally liked to use a more sophisticated video stream encoding, e.g., MPEG, the difficulty in obtaining a suitable codec motivated the use of streaming JPEG. Note however, that the conclusions of our experiment, which focuses on measuring adaptation agility, are themselves unaffected by the specific stream encoding

5.2 Effectiveness of Automatic Path Creation

To evaluate the effective of our path creation strategy, we experimented with it using the web access application, running under a wide range of network conditions.

CANS components used in the automatically generated access paths included: `ImageFilter` and `ImageResizer`, which are used to degrade image quality or resize JPEG images (to a factor of 0.2) respectively; `Zip` and `Unzip`, which work together to compress/decompress text. Together these components help the application overcome weak network links, albeit at a cost in node resources. The load and bandwidth factor values were obtained by profiling component execution on representative data inputs: a web page containing 14 KB text and six 25 KB JPEG images. In this experiment we used the same data inputs that the components were profiled on. This is a simplifying assumption, but reasonable given our primary focus here was to evaluate whether our approach could effectively adapt to multiple network conditions. Evaluating the effectiveness of the approach when component characteristics may be imprecise is deferred to Section 5.3.

To model different network conditions likely to be encountered along a mobile access path, we defined twelve different configurations listed in Table 1. These configurations represent the network bandwidth and node capacity available to a single client, and reflect different loading of shared resources and different mobile connectivity options.[4] These configurations are grouped into three categories, based on whether the mobile link L_2 exhibits cellular, infrared, or wireless LAN-like characteristics. Five of the configurations correspond to real hardware setups (tagged with a *), the remainder were emulated by restricting (via system call interception) CPU and network resources available to the application [3]. The computation power of different nodes is normalized to a 1 GHz Pentium III node.

Table 1 also identifies, for each platform configuration, the automatically generated plan for the web access application. The plans themselves are shown in Table 2, identifying the components that were automatically placed along the text and image paths. For example, plan A, automatically generated using the algorithm described in Section 3.2.1, places a `ImageFilter` and a `ImageResizer` on node N_1 along the image path, and a `Zip` and `Unzip` driver combination on nodes N_0 and N_2 along the text path.

Platform	*Edge Server* (N_0)	L_1	*Proxy Server* (N_1)	L_2	*Client* (N_2)	*Plan*
1	Medium	Ethernet	High	19.2 Kbps	Cell Phone	A
2	Medium	Ethernet	High	19.2 Kbps	Pocket PC	A
3^*	High	Fast Ethernet	Medium	57.6 Kbps	Laptop	B
4^*	High	Fast Ethernet	Medium	115.2 Kbps	Laptop	B
5	Medium	Ethernet	High	384 Kbps	Pocket PC	A
6^*	High	Fast Ethernet	Medium	576 Kbps	Laptop	B
7^*	Medium	Fast Ethernet	High	1 Mbps	Laptop	C
8	Medium	Ethernet	High	3.84 Mbps	Pocket PC	D
9	Medium	Ethernet	High	3.84 Mbps	Laptop	D
10	Medium	DSL	High	3.84 Mbps	Laptop	B
11	Medium	DSL	Low	3.84 Mbps	Laptop	B
12^*	Medium	Fast Ethernet	High	5.5 Mbps	Laptop	E

Relative computation power of different node types (normalized to a 1 GHz Pentium III node):
High = **1.0**, Medium = **0.5**, Laptop = **0.5**, Low = **0.25**, Pocket PC = **0.1**, Cell Phone = **0.05**
Link bandwidths:
Fast Ethernet = **100 Mbps**, Ethernet = **10 Mbps**, DSL = **384 Kbps**

Table 1: Twelve configurations representing different loads and mobile network connectivity scenarios, identifying the CANS plan automatically generated in each case.

Figure 5 shows the performance advantages of the automatically generated plans when compared to the response times incurred for direct interaction between the browser client and the server (denoted Direct in the figure). The bars in Figure 5 are normalized with respect to the best response time achieved on each platform. In all twelve configurations, the generated plans improve the response time metric, by up to a factor of seven. Note that part of the

[4]The bandwidth between the internet server and edge server available to a single client is assumed to be 10 Mbps.

Plan	N_0 (*Img/Txt*)	N_1 (*Img/Txt*)	N_2 (*Img/Txt*)
A	-/Zip	(Filter, Resizer)/-	-/Unzip
B	(Filter, Resizer)/Zip	-/-	-/Unzip
C	-/-	Filter/Zip	-/Unzip
D	-/Zip	-/-	-/Unzip
E	-/-	-/Zip	-/Unzip

Table 2: Component placement for the five automatically generated plans.

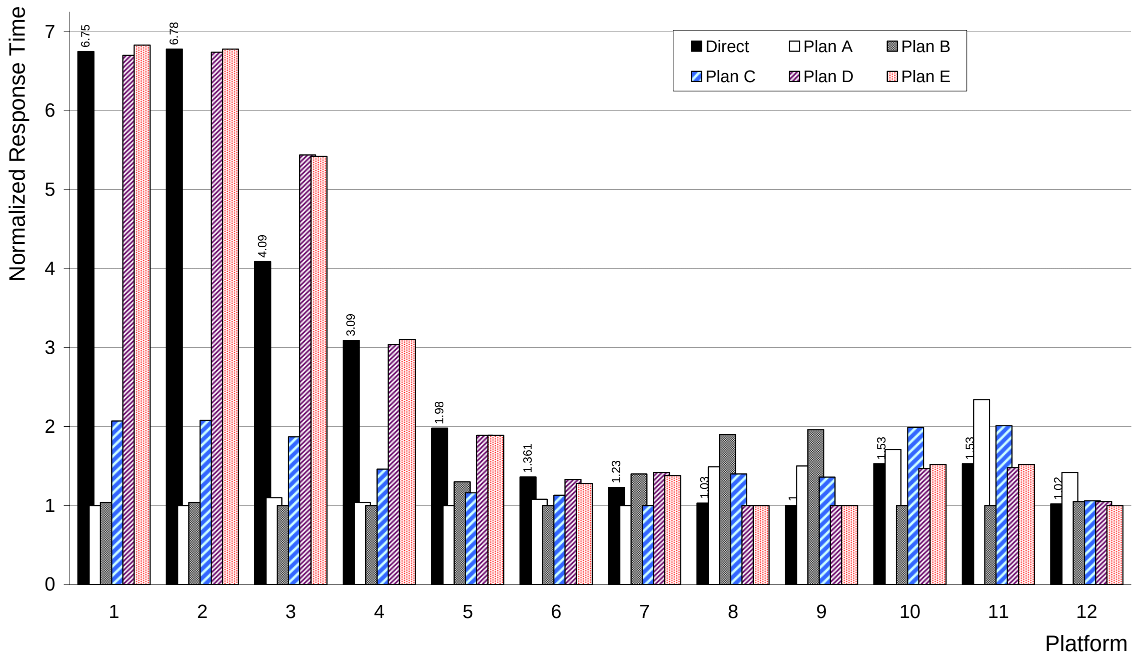


Figure 5: Response times achieved by different plans for each of the twelve platform configurations compared to that achieved by direct interaction. All times are normalized to the best performing plan for each configuration.

lower response times come at the cost of degraded image quality, but this is to be expected. The point here is that our approach *automates* the decisions of when such degradation is necessary.

Figure 5 also shows that different platforms require a different "optimal" plan, stressing the importance of automating the component selection and mapping procedure. In each case, the plan generated by our path creation strategy is the one that yields the best performance, also improving performance by up to a factor of seven over the worst-performing transcoding path. Note that while similar behavior can in principle be obtained by other strategies, such as using hard coded rules to deploy components. Unlike our application-neutral approach, such strategies require significant domain knowledge, and usually can not find the best path for network conditions that change continually.

Runtime Overhead To understand the run-time overheads incurred by flexible path-based approaches such as CANS, we profiled the application to construct a detailed timeline of operations. Our results, details of which are omitted here for space reasons, show that CANS incurs an average cost of 25μs per driver invocation, which is negligible for most applications.

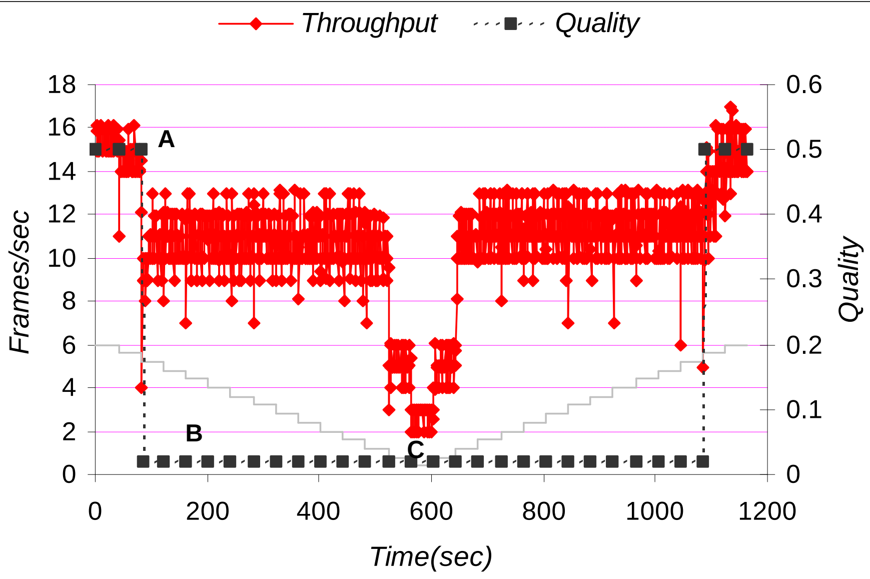
5.3 Effectiveness of Automatic Path Reconfiguration

To investigate the adaptation behavior of our approach in dynamic environments, we ran a set of experiments with the image streaming application. We began with the base planning mechanism, then applied the extension mechanisms (described in Section 3.2.2 and 3.2.3) one after another and quantified the improvement on the adaptation behavior exhibited. This set of experiments are used primarily to verify how fast CANS paths can respond to dynamic changes in network environments, and assume as stated earlier that an external module informs the path about resource availability changes. In practice, due to the unstable nature of shared wireless networks, this module must include a filtering mechanisms to determine whether a reconfiguration is required when a change is detected. We defer the construction of appropriate filters to our future work, noting only that other researchers have looked at similar issues [11].

The experiment modeled the following scenario: initially a user receives a bandwidth allocation of 150 KBps on the wireless link (L_2), which then goes down to 10 KBps in increments of 10 KBps every 40 seconds (modeling new user arrivals or movement away from the access point) before rising back to 150 KBps at the same rate (modeling user departures or movement towards the access point). The data path is allocated a (fixed) computation capacity of 1.0 (normalized to a 1 GHz Pentium III node) on nodes N_1 and N_2 respectively and a bandwidth of 500 KBps on L_1. N_1, N_2, and L_1 are wired resources and consequently more capable of maintaining a certain minimum allocation (e.g., by employing additional geographically distributed resources) than the wireless link L_2. The experiments were run on a wired-network with the wireless link behavior emulated as described earlier.

The components used in the image streaming example include the `ImageFilter` and `ImageResizer` introduced previously. To display incoming images appropriately, incoming throughput (frame rate) is required to be between 8 to 15 frames/sec. Within that range, better image quality is preferred.

Base Mechanism We started with the base planning strategy described in Section 3.2.1. Since it can only optimize one metric at a time, we chose to optimize throughput. The component parameters were obtained by profiling their behavior on a 25 KB JPEG image, one of a set of images ranging in size from 20–30 KB repeatedly transmitted by the server.



Plan	N_0	N_1
A	Filter	-
B	Filter	Resizer
C	Filter Resizer	-

Figure 6: Performance of CANS base mechanisms on the image streaming application.

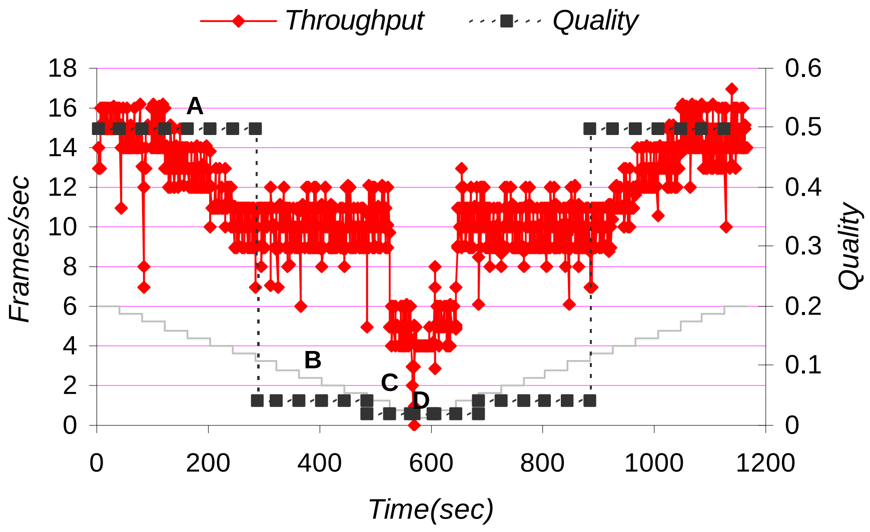
Figure 6 shows the throughput and image quality achieved by the data path over the 20 minute run of the experiment; the plans are shown in the table to the right. The plot needs some explanation. The light-gray staircase pattern near the bottom of the graph shows the bandwidth of link L_2 normalized to the throughput of a 25 KB image transmitted over the link; so, a link bandwidth of 150 KBps corresponds to a throughput of 6 frames/sec. The dashed black line corresponds to the quality achieved by the path. The jagged curve shows the number of frames received every second; because of border effects (a frame may arrive just after the measurement), this number fluctuates around the mean. The plateaus in the quality curve are labelled with the plan that is deployed during the corresponding time interval.

The results in Figure 6 show that the plans automatically created with our base mechanism do improve application throughput over what a static configuration would have been able to achieve. However, it also points out several deficiencies:

- Always trying to maximizing the throughput may sacrifice image quality unnecessarily, failing to meet application performance preferences.
- Inaccurate component parameters can cause unexpected throughput dropping (e.g., from plan A to B). The reason for this inaccuracy is that the `ImageResizer` in the data path processes filtered images while its behavior was profiled using original (unfiltered) images.

These two deficiencies are what motivated the extension of our planning algorithm to allow satisfaction of value ranges (Section 3.2.2), and the class-based profiling approach for more accurately modeling component behaviors. We describe the incremental benefits achieved by these extensions below.

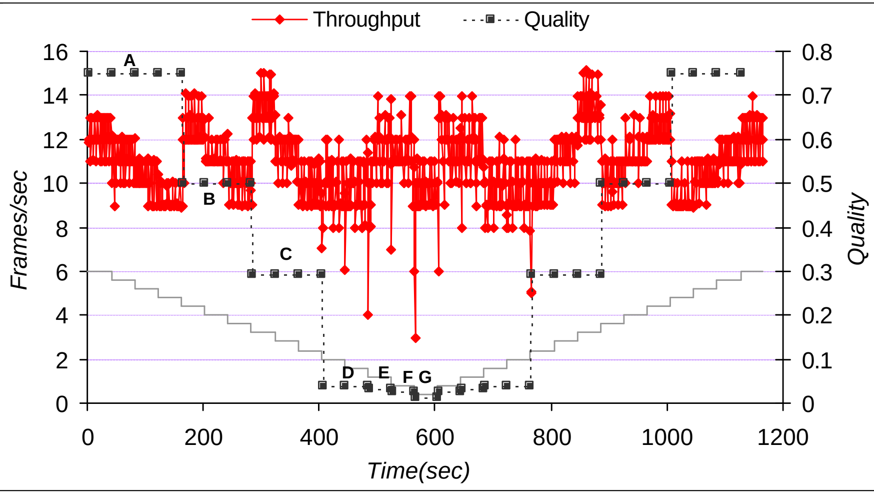
Planning for Value Ranges. After applying the range planning algorithm (Section 3.2.2), we obtained the result shown in Figure 7. Comparing with Figure 6, we can see two improvements. First, the range planning system retains Plan A for much longer than before (till 280 seconds into the experiment), choosing not to reconfigure while the throughput is still within the desired range. Second, the system employs an additional plan that falls between Plan A and B chosen in Figure 6 and represents a tradeoff that compromises on achieved throughput (while still ensuring that it is within the desired range) to improve quality. Such gradual decrease/increase in image quality is a desirable adaptation behavior expected by end users.



Plan	N_0	N_1
A	Filter	-
B	Resizer	-
C	Filter	Resizer
D	Filter Resizer	-

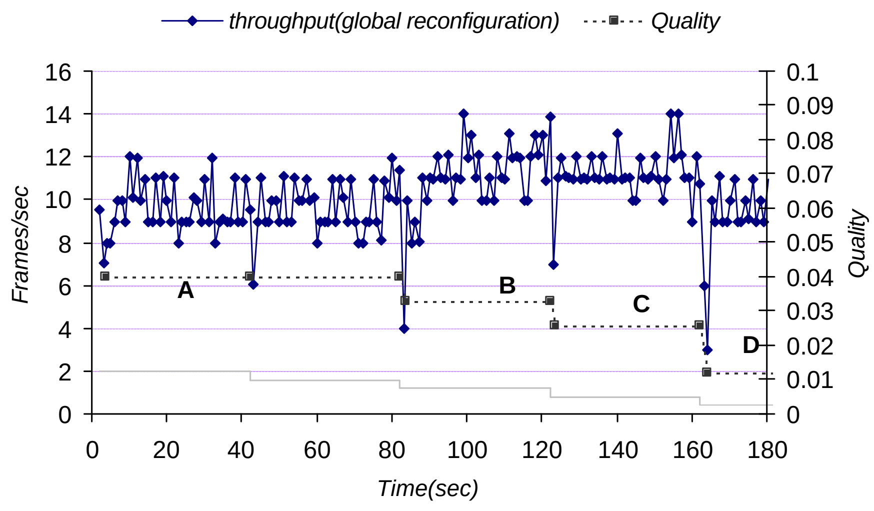
Figure 7: Performance with Range Planning Support

Refined Components Model . Observing the undesirable adaptations caused by inaccurate component parameters, we exploited the *class profiling* technique described in Section 3.1. In addition, we also allowed both components in our image streaming application to support multiple configurations: nine configurations for `ImageFilter` with quality value ranging from 0.1 to 0.9, four configurations for `ImageResizer` with scale factors ranging from 0.2 to 0.8. The components were profiled for three different image classes (high, medium, low). Figure 8 shows the resulting performance and associated plans. There are three obvious improvements over Figure 7. First, the throughput is kept in the required range for the whole duration of the experiment (except for transition points caused by reconfigurations). Second, the image quality changes more smoothly than in Figure 7. Instead of 3 configurations (quality levels), there are 7 different plans, resulting in smoother variations in path quality. Finally, the low costs of switching configurations is reflected in transitions from plans A to B, and B to C, which hardly disrupt the achieved throughput unlike the associated cost for introducing a new component (e.g. transition between plans C and D). The large number of automatically selected plans, which are required for satisfying the application preferences are yet another indication of the benefits of an automatic approach: accomplishing similar behaviors using a hard-coded approach would necessitate detailed domain knowledge and comprehensive involvement of application developers.



Plan	N_0	N_1
A	Filter(4)	-
B	Filter(3)	-
C	Filter(1)	-
D	-	Resizer(1)
E	Resizer(1)	Filter(5)
F	Resizer(1)	Filter(3)
G	Resizer(1)	Filter(1)

Figure 8: Performance with Multi-Configuration Components and Class Profiling



Plan	N_0	N_1
A	-	Resizer(1)
B	Resizer(1)	Filter(5)
C	Resizer(1)	Filter(3)
D	Resizer(1)	Filter(1)

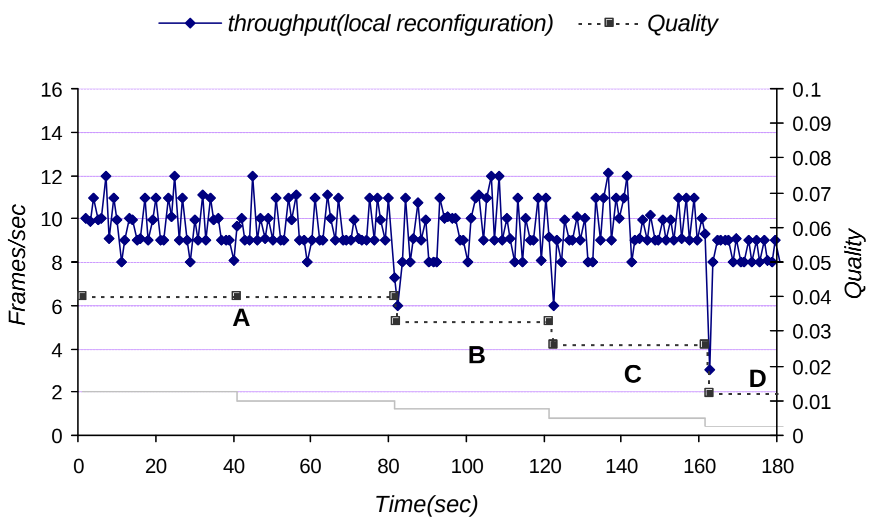
Figure 9: Performance of Global Reconfiguration

5.4 Overhead of Path Reconfiguration

To understand the cost of path reconfiguration, and to investigate the difference in adaptation behavior using global and local reconfiguration mechanisms, we revisited the experiments using the image streaming application, focusing on the segment when bandwidth dropped from 50KBps to 10KBps. We ran the experiment in two ways: with local reconfiguration turned off (i.e., only global reconfiguration was enabled) and with it turned on.

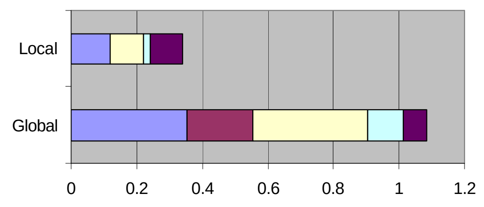
Throughput measured for the global and local cases are shown in Figure 9 and Figure 10 respectively. Unlike global reconfiguration, which partitions the `ImageResizer` and `ImageFilter` portions of the data paths in plans B, C, and D, so that they run on both nodes N_0 and N_1 to obtain a slightly higher value of throughput, local reconfiguration strategy chooses to both calculate the plan and deploy the components on the same node, thereby avoiding the coordination cost across two nodes. As a result, local reconfigurations produce more stable throughput during reconfiguration (look at the reconfiguration that happens at the 80^{th} second). The cost however is that the local reconfiguration does not quite achieve the same throughput as the global case, achieving 10 frames/sec instead of 12. Note that this is still within the desired range, otherwise global reconfiguration would have been triggered.

A cost breakdown of the reconfigurations at the 80^{th} second point for the two cases is shown in Figure 11. The cost is broken down into five stages corresponding to the steps described in Section 4.2: planning, shipping the computed plan to relevant nodes, flushing any data in the reconfigured segment, inserting the components for the new plan, and



Plan	N_0	N_1
A	-	Resizer(1)
B	-	Resizer(1),Filter(5)
C	-	Resizer(1),Filter(3)
D	-	Resizer(1),Filter(1)

Figure 10: Performance of Local Reconfiguration



	Global	Local
🟪 1st Message	0.07	0.1
▢ Reconfiguration	0.11	0.02
▢ Flush-Data	0.35	0.1
🟥 Partition-Shipping	0.2	0
🟦 Planning	0.354	0.12

Figure 11: Reconfiguration Cost

resuming data transmission. Global reconfiguration requires 1.08 seconds to complete as compared to 0.35 seconds for the local case. The major portion of this difference, which stems from additional network communication required in the global case, is restricted to the first three stages, verifying our claim that local reconfiguration can substantially improve responsiveness. Note that in both the local and global cases, data transmission is only suspended during the last two stages (0.18 seconds and 0.12 seconds in the global and local cases), so most applications can continually adapt to dynamic changes.

5.5 Summary of Results

The experiments described in this section verify that (1) network-aware data paths can be automatically created and dynamically configured; (2) these automatically created paths outperform others and provide considerable performance advantage for applications by matching underlying network conditions, while incurring negligible runtime overhead;(3) our mechanisms can provide applications with fine tuned, desirable adaptation behaviors; and that (4) path reconfiguration cost is small for most applications, which can be further substantially reduced with supported local operations.

6 Discussion

6.1 Comparison with other automatic path creation approaches

Automatic path creation approaches have also been proposed in the context of other general path-based frameworks, such as Ninja [8], Scout [16], and Conductor/Panda [19].

The Ninja project's Automatic Path Creation (APC) service [8] deploys components at proxy sites (active proxies) to distill/transform data to suitable formats for different types of end devices. Though both APC and CANS formulate the component selection problem in terms of type compatibility, the perspectives of Ninja and our approach are very different. The primary focus of Ninja APC is to address the diversity in capacity of end devices and last hop links. Our approach takes a more general view that network resource conditions at each part of network paths can be different, and that more importantly, these conditions can change dynamically because of client load or other reasons. As a result, paths created with Ninja APC are basically function-oriented without further optimization for performance, essentially offering differentiated service access according to a small number of classes. Paths built using our approach are performance-oriented and have built-in support for reconfiguration, two features that are not supported in Ninja APC.

Recent work in the Scout project [16] has looked at a template based path construction algorithm for delivering media objects that takes into consideration resource requirements, user preferences, and node capabilities. Unlike our approach, the Scout algorithm requires a database of predefined path templates, simply instantiating an appropriate template based on other programmer-provided rules that decide whether or not a component can be created on a resource. As our experimental results show, such template-based approaches rely on a significant amount of domain knowledge that may or may not be appropriate for network resources that can change continually. Reconfiguration is not supported in this work.

The Panda project [19] also proposes a planning scheme for placing network-level components to modify an application's data stream in response to unfavorable network conditions. Two schemes are discussed, one based upon selection from a reusable plan set and the other based on exhaustive constraint space-based search. The reusable plan set approach, similar to the template database used in Scout, cannot guarantee the optimality of generated paths, especially for continually changing resources. The latter approach does not scale well when the total number of components increase to a large amount (for example, there may exist many different components to do similar operations such as compression or encryption). To the best of our knowledge these schemes have not yet been implemented or evaluated with real applications. Panda also provides error recovery mechanisms for reliable data transmission, but does not provide support for dynamic reconfiguration. Besides, unlike our approach, the error recover mechanisms in Panda require complete knowledge of the whole path, which may result in long recovery/transition times.

Some projects investigating multimedia content delivery have also proposed the processing or customization of stream content along network paths. In [24], an approach is proposed to find a safest path (by mapping a sequence of processing operators) on media service proxy network to minimize the possibility of failing to deliver the content. Though resource availability is considered in this work, such paths do not provide optimized performance, and dynamic adaptation is not supported. Furthermore, since the approach is designed for multimedia content delivery, the selection of components benefits from more domain knowledge than general application-neutral path-based approaches.

The primary differences of our approach as compared to these infrastructures is its emphasis on application-neutral mechanisms for path creation and reconfiguration, which are capable both of optimizing the performance of data paths to encountered network conditions, and dynamically adapting these paths as conditions change. Although much work remains, to the best of our knowledge, CANS is the first path-based approach to exhibit both of these properties.

6.2 Plan for future work

Our research using the CANS infrastructure has currently focused on only a core set of concerns, specifically the feasibility and performance of automatic path creation and reconfiguration strategies.

A complete solution to automatically building network-aware access paths from application-level components requires addressing several other equally important issues, which we are investigating in current work.

- In general, paths generated by CANS-like approaches may need to span multiple network administrative domains, which introduces two challenges. First, there is a need for mechanisms such as dRBAC [6] that make explicit the trust relationships between domains, and thereby influence which components can be deployed where in the network. Second, the component and type model in CANS needs to be extended to cope with the fact that not all components are available for deployment outside a particular domain, and that the same semantic type may be referred to differently across the domains. The good news for the latter issue is that our dynamic programming planning algorithm can be easily extended to allow each domain to do planning for the path segment within it. But there is still a need for agreement on the types that are used across domain boundaries.

- Our current work has assumed the presence of external modules responsible for resource monitoring and allocation of available resources among multiple paths. While these problems are being investigated by other researchers, more work is needed before a robust solution to these problems is available. Allowing paths to span multiple domains in particular introduces the problem that resource monitoring information needs to integrated from multiple distributed sources (e.g., [2, 12, 13]). Additionally, protocols that efficiently divide up available resources among multiple concurrently active paths must tradeoff fairness issues against increased utilization.

6.3 Limitation(s) of CANS-like approaches

CANS components are specified in terms of their input and output data types, and hence CANS is most suitable for deploying components that transform the format of data presentation while retaining the same underlying semantic for transmitted data. For example, a transcoded web page still represents the original. While this is a limitation, many current-day mobile service access applications can benefit from components that do fall into this category: filtering, caching, protocol conversion, and transcoding etc.

Automatic deployment of semantics-transforming components would require going beyond our type model, possibly leveraging standard ontologies such as being developed by the Semantic Web [1] and IEEE's Standard Upper Ontology [9] efforts. Note that our approaches can also be used with these enhanced mechanisms, which can be thought of as generalizing the notion of type compatibility.

7 Conclusions

This paper has presented and evaluated an automatic approach for the dynamic construction and reconfiguration of network-aware paths across networks. In contrast to other path base approaches, our work proposed a model and corresponding algorithms to build network paths whose performance is optimized for matching underlying network conditions, and efficient reconfiguration mechanisms for these paths to be dynamically and continually reconfigured to satisfy user preferences and network resource constraints. The creation and reconfiguration of paths are accomplished *automatically* with minimal input from applications. Although much work remains, to the best of our knowledge, these results are among the first to demonstrate the viability of using fully automatic path-based adaptation approaches.

References

[1] Tim Berners-Lee. Services and semantics: Web architecture. In *http://www.w3.org/2001/04/30-tbl.html*, 2001.

[2] P. Chandra, A. Fisher, C. Kosak, T. S. Eugene Ng, P. Steenkiste, E. Takahashi, and H. Zhang. Darwin: Resource management for value-added customizable network service. In *Sixth IEEE International Conference on Network Protocols (ICNP'98)*, October 1998.

[3] F. Chang, A. Itzkovitz, and V. Karamcheti. User-level Resource-Constrained Sandboxing. In *Proc. of the 4th USENIX Windows Systems Symposium*, August 2000.

[4] A. T. Campbell et al. A Survey of Programmable Networks. *ACM SIGCOMM Computer Communication Review*, April 1999.

[5] A. Fox, S. Gribble, Y. Chawathe, and E. A. Brewer. Adapting to Network and Client Variation Using Active Proxies: Lessons and Perspectives. *IEEE Personal Communication*, September 1998.

[6] E. Freudenthal, T. Pesin, E. Keenan, L. Port, and V. Karamcheti. drbac: Distributed role-based access control for dynamic coalition environments. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, July 2002.

[7] X. Fu, W. Shi, A. Akkerman, and V. Karamcheti. CANS:Composable, Adaptive Network Services Infrastructure. In *Proc. of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS)*, March 2001.

[8] S. D. Gribble and et al. The Ninja Architecture for Robust Internet-Scale Systems and Services. *Special Issue of IEEE Computer Networks on Pervasive Computing*, 2000.

[9] Standard Upper Ontology (SUO) Working Group. Ieee standard upper ontology scope and purpose. In *http://suo.ieee.org/scopeAndPurpose.html*, 2001.

[10] A. D. Joseph, J. A. Tauber, and M. F. Kasshoek. Mobile Computing with the Rover Toolkit. *IEEE Transaction on Computers:Special Issue on Mobile Computing*, 46(3), March 1997.

[11] M. Kim and B. Noble. Mobile network estimation. In *Proceedings of the Seventh ACM Conference on Mobile Computing and Networking*, July 2001.

[12] K. Lai and M. Baker. Nettimer: A tool for measuring bottleneck link bandwidth. In *Proc. of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS)*, March 2001.

[13] B. Lowekamp, N. Miller, D. Sutherland, T. Gross, P. Steenkiste, and J. Subhlok. A resource query interface for network-aware applications. In *7th IEEE Symposium on High-Performance Distributed Computing*, July 1998.

[14] A. Mallet, J. Chung, and J. Smith. Operating System Support for Protocol Boosters. In *Proc. of HIPPARCH Workshop*, June 1997.

[15] R. Mohan, J. R. Smith, and C.S. Li. Adapting Multimedia Internet Content for Universal Access. *IEEE Transactions on Multimedia*, 1(1):104–114, March 1999.

[16] A. Nakao, L. Peterson, and A. Bavier. Constructing End-to-End Paths for Playing Media Objects. In *Proc. of the OpenArch'2001*, March 2001.

[17] Brian D. Noble. *Mobile Data Access*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1998.

[18] G. Parks. 802.11e makes wireless universal. *Available at http://www.nwfusion.com/news/tech/2001/0312tech.html*, March 2001.

[19] P. Reiher, R. Guy, M. Yavis, and A. Rudenko. Automated Planning for Open Architectures. In *Proc. of OpenArch'2000*, March 2000.

[20] sharewave.com. http://www.sharewave.com/.

[21] P. Sudame and B. Badrinath. Transformer Tunnels: A Framework for Providing Route-Specific Adaptations. In *Proc. of the USENIX Technical Conf.*, June 1998.

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

[23] D. J. Wethrall, J. V. Guttag, and D. L. Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols. In *Proc. of 2nd IEEE OPENARCH*, 1998.

[24] Dongyan Xu and Klara Nahrstedt. Finding service paths in a media service proxy network. In *Proc. of SPIE/ACM Conf. on Multimedia Computing and Networking (MMCN 2002)*, Jan 2002.

[25] M. Yavis, A. Wang, A. Rudenko, P. Reiher, and G. J. Popek. Conductor: Distributed Adaptation for complex Networks. In *Proc. of the Seventh Workshop on Hot Topics in Operating Systems*, March 1999.