Planning for Network-Aware Paths

Xiaodong Fu and Vijay Karamcheti

New York University, New York, NY 10012, USA,
{xiaodong,vijayk}@cs.nyu.edu

Abstract. Communication in distributed applications across a wide area network needs to cope with heterogenous and constantly changing network conditions. A promising approach to address this is to augment the *whole* communication path with *network awareness* by 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 automatically creating effective network paths whose performance is optimized for encountered network conditions.

This paper describes a solution for this problem. Our approach, which is built into an application-level programmable network infrastructure called CANS (Composable Adaptive Network Services), constructs network-aware communication paths that enhance application performance by taking into account both application performance preferences and dynamic resource availability.

Our experiments with typical applications verify that communication paths automatically created with our path creation algorithms do bring applications with considerable performance advantages, and fine tuned, desirable adaptation behaviors, with only minimal input from applications.

1 Introduction

Heterogeneous and dynamically changing network environments are an important cause for unsatisfactory behaviors of distributed applications whose performance is directly related to the quality of the underlying data communication. A promising approach ([2, 4, 8, 9, 11–13, 15]) for addressing this is to make the communication path *network aware* by using application-specific components that handle stream degradation, reconnection, transcoding, caching, and protocol conversion operations, thereby serving to “impedance match” application performance requirements with the underlying network conditions. The more general among these infrastructures [4, 9, 12, 13, 15] propose to realize such network awareness throughout *the whole communication path*. While there have been a large number of proposals, such *path-based* systems have focused primarily on providing system support to allow dynamic insertion and deletion of components, leaving unanswered a key question: how to automatically construct such network aware paths, without user involvement, so that applications can perform better in a dynamic network environment where resource availability changes continually.

In this paper, we propose an automatic strategy for building such network aware paths with optimized performance in accordance with application performance requirements and underlying resource availability. In addition to calculating a whole path, our solution can also be used with disjoint segments of an existing path independently while maintaining some overall performance guarantee. Furthermore, the calculation of communication paths can be conducted in a distributed fashion (i.e. from one network domain to another). These properties make our approach applicable in a wide area setting, where multiple network domains are usually involved in setting up and maintaining a communication path.

This strategy is a general solution that can be applied to any path based system. To evaluate it, we have implemented it within a programmable network infrastructure called CANS (Composable Adaptive Network Services) [4], and have conducted a series of experiments, 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 is achievable and does in fact yield substantial performance benefits; and that (2) our approach is effective for providing applications that have different performance preferences with fine tuned, desirable adaptation behaviors.

The rest of this paper is organized as follows. Section 2 provides a brief overview of the CANS infrastructure. Section 3 defines the path creation problem and our model. Section 4 describes our automatic path creation algorithm and extensions. Section 5 evaluates these mechanisms using the two applications. Section 6 reviews related work and summarizes the novel aspects of our approach. We conclude in Section 7.

2 Background: Overview of the CANS Architecture

The **C**omposable **A**daptive **N**etwork **S**ervices (CANS) infrastructure [4] views network environments as consisting of client *applications* and *services*, connected by *communication paths*. CANS extends the notion of a communication path, traditionally limited to data transmission, to include dynamically injected application-specific components. Serving as the basic building block for CANS paths, components are standalone *mobile* code modules that can be composed via a standard *data port* interface.

The CANS network is realized by partitioning service and 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. CANS further provides support for dynamic path reconfiguration. A more detailed description of the CANS infrastructure can be found in [4].

3 Modeling the Path Creation Problem

In general, creation of a network aware 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, ISP-level agreements, etc.) and so here we focus only on the problem of component selection and mapping. We call the procedure of constructing such paths as *planning*.

3.1 Components and Network Resources

We first need ways of characterizing the impact of a particular component on the resource utilization along a path as well as a means for associating performance metrics with the overall path.

Types The functionality of a component in a path is modeled as transforming data from one type to another. For example, a compression component using the zip algorithm can transform a MIME/TXT type to a “zipped” MIME/TXT type. Components can be connected together only if their type information is compatible. Type compatibility is defined in a **type graph** G_t: a vertex in the graph represents a type, and an edge represents a component that can transform data from the source type to the sink type. The primary benefit of such type-based modeling is that it permits description of valid candidate paths without explicit enumeration, simply looking for all type-compatible sequences of components that transform the source type to the required destination type.

The performance of an individual path is determined by resource availability and performance characteristics of components in the path.

Network resources Each network resource is modeled in terms of its performance characteristics, i.e. computation capacity for a node, and bandwidth and latency for a network link. For an individual path that passes through a shared network resource, the value used is the corresponding value of the allocated portion.

Furthermore, network resources may also affect the data passing through them in a way that is independent of resource capacities. For example, the effect is different for transmitting sensitive data across a network link with the same bandwidth/latency parameters, depending on whether the link is trusted (no eavesdropper) or not. To model such effects, CANS incorporates a notion of *augmented types* [4], which extend data types with environment properties. Network resources are modeled as entities that can transform the environment properties of augmented types in a type-specific fashion.

Modeling both application data types and resource constraints using a unified framework has the advantage that valid paths (even in the presence of resource constraints) continue to be concisely represented 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 component, each component c is modeled in terms of its *computation load factor* ($\mathrm{load}(c)$), the average per-input byte cost of running the component, and its *bandwidth impact factor* ($\mathrm{bwf}(c)$), the average ratio between input and output data volume. This simple model can be extended to allow components to have multiple configurations.

A path ($D = \{c_1, \ldots, c_n\}$) consists of a sequence of components. A **route** $R = \{n_1, n_2, \ldots, n_p\}$ for a path is a sequence of network resources (nodes and links between them). A **mapping**, $M : D \to R$, associates components on data path D with nodes in route R. We are only interested in mappings that satisfy the following restriction: $M(c_i) = n_u, M(c_{i+1}) = n_q \Rightarrow u \leq q$: sending data back and forth between nodes in a route usually results in poor performance and resource waste.

3.2 Problem Definition

The path creation problem can be stated as the following: given a route R (with resources allocated to the path), a type graph G_t, a source data type t_s, a destination data type t_d, select a data path D that 1) transforms t_s to t_d and can be mapped to R, and 2) provides optimal performance (e.g. maximum throughput or minimal latency).

4 Algorithm

Our path creation strategy, in addition to satisfying type requirements, respects constraints imposed by node and link capacities and properties and optimizes some overall path metric (e.g., latency or throughput). The heart of our strategy is a dynamic programming algorithm, which simultaneously selects and maps a type-compatible component sequence to optimize some performance metric.

We first describe a base version of the algorithm in which a single performance metric needs to be optimized. We then present an extension for applications that require the value of some performance metric to be in an *acceptable range*. For such applications, 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); 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. Lastly, we describe a distributed implementation of this strategy.

4.1 Base Algorithm

Unfortunately, finding the optimal solution for the path creation problem defined in Section 3.2 is an NP-hard problem. The complexity mainly comes from the large numbers for both components and the possible ways to compose them, as well as different ways to map them to network resources.

However, this problem can be made tractable with a reasonable simplification: we partition the computation capacities of nodes 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*, $\boldsymbol{A}(R) = (r_1, r_2, \ldots, r_p)$, 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 maximum throughput as the goal of performance optimization (other performance metrics can also be used); we use p to denote 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.

Dynamic Programming Strategy

The intuition behind the algorithm is to incrementally construct, for different amounts of route resources, optimal mappings with increasing numbers of components, say $i+1$, using as input optimal partial solutions involving i or fewer components.

To construct a solution with $i + 1$ (or fewer components) for a given destination type t and resource vector $\boldsymbol{A}$, we consider all possible intermediate types t' that can be transformed to t; i.e., all those types for which an edge (t', t) is present in the type graph. For each such t', we consider all possible mappings of the associated component c on nodes along the route that use no more than $\boldsymbol{A}$ resources. For each such mapping that transforms the available resource vector to $\boldsymbol{A'}$ (after accounting for $\mathrm{load}(c)$), we combine this component with the previously calculated solution for t' with i (or fewer) components with resource vector $\boldsymbol{A}$. The combined mapping that yields the maximum throughput is deemed the solution at step $i + 1$ for type t.

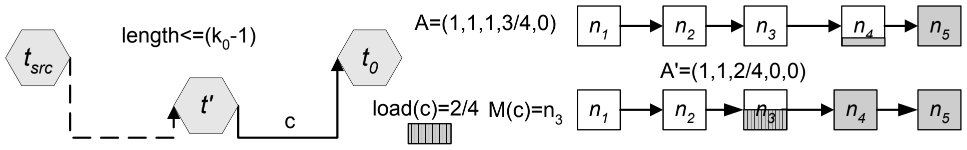


Fig. 1: Map c to n_3 and lookup solution with $\boldsymbol{A'}$.

```
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 $t, \boldsymbol{A} \in \mathrm{RA}$
3.      do calculate $s[t_s, t, \boldsymbol{A}, 0]$
4.  (∗ Step 2: Incrementally building partial solutions ∗)
5.  for $i \leftarrow 1$ to $p \times n$
6.      do for all $t \in V(G_t), \boldsymbol{A} \in \mathrm{RA}$
7.              do $s[t_s, t, \boldsymbol{A}, i] \leftarrow s[t_s, t, \boldsymbol{A}, i-1]$
8.                  for all $c = (t', t) \in E(G_t)$
9.                      do for all $n_j$ that $\boldsymbol{A}[n_j] > 0$
10.                             do $M(c) \leftarrow n_j$
11.                                 $\boldsymbol{A'} \leftarrow (\boldsymbol{A}[0], \ldots, \boldsymbol{A}[n_j - 1], \boldsymbol{A}[n_j] - \mathrm{load}(c), 0, \ldots)$
12.                                 $\mathrm{TH} \leftarrow \mathrm{throughput}(\mathrm{append}(s[t_s, t', \boldsymbol{A'}, i-1], c, \boldsymbol{A}))$
13.                                 if $\mathrm{TH} > s[t_s, t, \boldsymbol{A}, i]$
14.                                   then $s[t_s, t, \boldsymbol{A}, i] \leftarrow \mathrm{TH}$
15. return $s[t_s, t_d, \boldsymbol{A} = [1, 1, ..., 1], p \times n]$
```

Fig. 2: Base Path Creation Algorithm

Because this procedure runs backwards from the destination (i.e. c_{j+1} is mapped before c_j), consequently, only resource vectors of the form $(1, ..., 1, r_j \in [0, 1], 0, ..., 0)$ will be used in the calculation. These set of resource vectors is designated RA. The size of RA is $O(p)$.

Formally, the algorithm fills up a table of partial optimal solutions ($s[t_s, t, \boldsymbol{A}, i]$) in the order $i = 0, 1, 2, \ldots$. The solution $s[t_s, t, \boldsymbol{A}, i]$ is the data path that yields maximum throughput for transforming the source type t_s to type t, using i or fewer components and requiring no more resources than $\boldsymbol{A}$($\boldsymbol{A} \in \mathrm{RA}$). Figure 1 shows the moment in the calculation of $s[t_s, t_0, (1, 1, 1, 3/4, 0), i + 1]$ when the the component c is mapped to node n_3, and appended with partial solution $s[t_s, t', (1, 1, 2/4, 0, 0), i]$. Note that in this example, computation capacity of nodes is partitioned into 4 intervals.

The algorithm terminates at Step $p \times n$, with the solution in $s[t_s, t_d, (1, ..., 1), p \times n]$. 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 $O(n^2 \times m \times p^3) = O(n^3 \times p^3)^1$ as opposed to $O(p^n)$ for an exhaustive enumeration strategy. n, the total number of components, usually is a big number. Even for a simple operation, such as compression, there may exists many different candidates, not to mention that each component may have multiple configurations. Therefore, $O(p^n)$

[1] It is safe to assume that $m < n$.

is infeasible in practice. In most scenarios, p is expected to be a small constant, therefore overall complexity of our path creation algorithm is determined by the number of components. The pseudo code of this algorithm is shown in Figure 2.

4.2 Extension: Planning for Value Ranges

Given that our planning algorithm constructs communication paths by incrementally filling in a solution table of $s[t_s, t, \boldsymbol{A}, i]$, 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 (e.g. appending compression components to a partial path can bring down overall latency). To *estimate* whether the desired range can in fact be 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 meet the requirement using residual resources along a path that transforms type t to t_d. Using this information, when calculating $s[t_s, t, \boldsymbol{A}, i]$, those partial solutions that can not meet the requirement will be discarded in the first place. Heuristic functions are used for choosing among candidate paths that can all meet the required range. Note that complementary planning needs to be run just once.

Planning for value ranges can further be extended to calculate plans for a portion of the whole path. Such a local mechanism allows disjointed segments of a data path to change their behaviors independently and concurrently while maintaining some overall performance guarantee. Due to space limitations, we omit the details about the local planning mechanism, which can be found in a technical report [3].

4.3 Distributed (Incremental) Planning.

Though our path creation strategy has so far been described in a centralized manner, it can easily be extended to run in a distributed fashion. To do that, each node (n_i) on the route just needs to calculate $s[t_s, t, \boldsymbol{A} = (\underbrace{1, ..., 1}_{i}, 0, ..., 0), \sum_{j=0}^{i}(\mathrm{CN}_j)]$ (where CN_j is the total number of components in node n_j), and send these partial solutions to the next node. This procedure starts from the server node and continues until it reaches the client node.

The primary benefit of this distributed version is that there is no need for a centralized planner that has a complete knowledge of components and types for all nodes in the route. By incrementally calculating a path in such a distributed fashion, only knowledge for common types that are used across different network domains is necessary. This distributed version, combined with the local mechanisms described earlier, enables a path-based system to be used in a wide area network, where a communication path usually spans multiple administration domains.

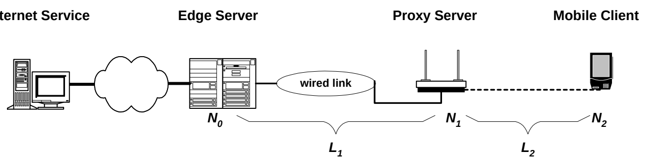


Fig. 3: A typical network path between a mobile client and an internet services.

The traffic incurred for the distributed planning is just messages of partial solutions between adjacent nodes. It should be noted here that these messages carry only values of the performance metric, transmission of components is unnecessary.

5 Performance Evaluation

To evaluate the effectiveness of our approach, we have built the automatic path creation support into the CANS infrastructure, and conducted a series of experiments in the context of a web access application and an image streaming application.

5.1 Experimental Platform

We consider a typical network path between a mobile client and an Internet server as shown in Figure 3. This platform models a mobile user using a portable device (N_2) 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: N_0, N_1, and N_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 JMF application that continuously fetches JPEG frames from an image server and displays them. To perform appropriately, this application requires a certain frame rate, and prefers high data quality.

Components used in the automatically generated 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. The load and bandwidth factor values, which are omitted here for brevity, were profiled using representative data inputs: a web page containing 14 KB text and six 25 KB JPEG images.

5.2 Effectiveness of the Base Path Creation Algorithm

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

In particular, 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

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.

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.

options.[2] 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 [1]. 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 image and text paths. For example, plan A, which is used in platform 1 and 2, 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.

Figure 4 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 4 are normalized with respect to the best response time achieved on each platform. In all twelve configu-

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

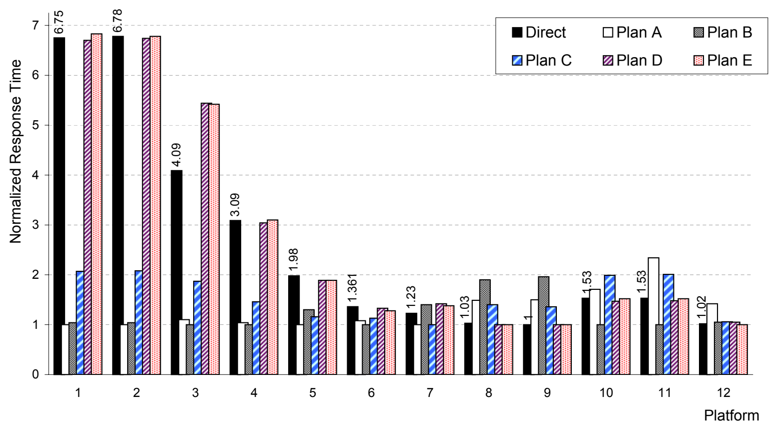


Fig. 4: Response times achieved by different plans for each of the twelve platform configurations compared to that achieved by direct interaction. All times are normalized to the best performing plan for each configuration.

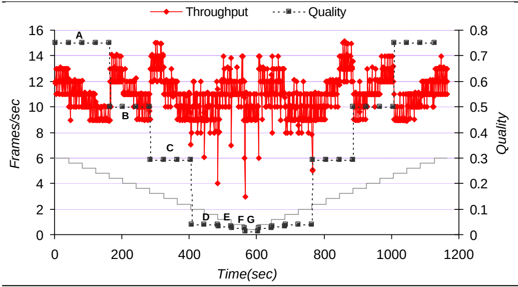
rations, the generated plans provide the best performance, improving the response time metric by up to a factor of seven. Note that part of the lower response times come at the cost of degraded image quality, but this is to be expected. The point here is that our approach *automates* the decisions of when such degradation is necessary.

Figure 4 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 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 cannot find the best path for network conditions that change continually.

5.3 Planning for Value Ranges

Unlike the web accessing application, the image streaming application requires throughput of the data path to be in a particular range so that the received data can be appropriately rendered on the client devices. To validate our range planning strategy and further investigate the adaptation behavior achieved using our approach in dynamic environments, we experimented with this application.

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



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)

Fig. 5: Performance of the Image Streaming Application

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 by controlling available bandwidth of the application via system call interception [1], as in some configurations in the web access experiment. In this experiment, an external module was used to inform the path about resource availability changes.[3]

The components used in the image streaming example include the `ImageFilter` and `ImageResizer` introduced previously. In addition, we also allowed both components in our image streaming application to support multiple configurations: nine configurations for `ImageFilter` with quality values ranging from 0.1 to 0.9, four configurations for `ImageResizer` with scale factors ranging from 0.2 to 0.8. 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.

Figure 5 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 5 show that the plans automatically created with our range mechanism do provide desirable adaptation behavior. First, the throughput is kept in the required range for the whole duration of the experiment (except for transition points caused by reconfigurations). Second, as a result of our range planning strategy, the image quality is decreased gradually, resulting in smooth variations in path quality. Using optimization solely for throughput will select paths with more compression capability,

[3] In practice, due to the unstable nature of shared wireless networks, this module must include filtering mechanisms to determine whether a reconfiguration is in fact 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 [7].

resulting in unnecessarily high throughput at the expense of worse image quality. 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.

6 Related Work

Our work is related to previous work on constructing network aware communication paths.

The Ninja project's Automatic Path Creation (APC) service [5] deploys components at proxy sites (active proxies) to distill/transform data to suitable formats for different types of end devices. 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 the network path can be different, and more importantly, these conditions can change continually. 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 in the sense that they can provide applications with performance optimized for the conditions.

Template-based or reusable plan sets are used in the Scout [9] and Panda projects [10]. Unlike our approach, these approaches require a database of predefined path templates (or reusable plan sets), 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 would need to rely on a significant amount of domain knowledge that may or may not be appropriate for network resources that can change continually.

Recent work on multimedia content delivery [14] has also proposed an approach to find a safest path (by mapping a sequence of processing operators) on a 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. 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 same path construction problem exists in service composition across a wide area network with QoS requirements. Recent work [6] has proposed the use of heuristic strategies to map a given sequence of service instances for required QoS parameters. Differing from this work that focuses only on the mapping of service instances, our approach solves component selection and mapping as a combined problem. Dividing the component selection and mapping into two separate stages may exclude valid solutions and impair the optimality of the produced path.

7 Conclusions

This paper has presented a model and corresponding algorithms for building network-aware communication paths whose performance is optimized for the underlying network conditions. Though built in the CANS infrastructure, our approach is applicable for all general path-based systems that aim to provide applications with support for adapting to dynamic changes in the network. The experiment results validate our approach, showing that the paths created automatically with our approach not only bring applications considerable performance benefits, but also provides desirable adaptation behaviors, requiring only minimal input from the applications. Furthermore, our algorithm can be applied to disjoint segments of a path independently and can be calculated in a distributed fashion, thus making it suitable for being used in a wide area network.

Acknowledgements We'd like to thank Weisong Shi for his help with experiments using the web access application. This research was sponsored by DARPA agreements N66001-00-1-8920 and N66001-01-1-8929; by NSF grant CAREER: CCR-9876128 and CCR-9988176; and Microsoft.

References

1. F. Chang, A. Itzkovitz, and V. Karamcheti. User-level Resource-Constrained Sandboxing. In *Proc. of the 4th USENIX Windows Systems Symposium*, August 2000.
2. 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.
3. X. Fu and V. Karamcheti. Automatic creation and reconfiguration of network-aware service access paths. Technical Report TR2002-824, New York University, March 2002.
4. 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.
5. 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.
6. X. Gu, K. Nahrstedt, R. N. Chang, and C. Ward. Qos-assured service composition in managed service overlay networks. In *Proceedings of The 23rd International Conference on Distributed Computing Systems*, May 2003.
7. M. Kim and B. Noble. Mobile network estimation. In *Proceedings of the Seventh ACM Conference on Mobile Computing and Networking*, July 2001.
8. A. Mallet, J. Chung, and J. Smith. Operating System Support for Protocol Boosters. In *Proc. of HIPPARCH Workshop*, June 1997.
9. A. Nakao, L. Peterson, and A. Bavier. Constructing End-to-End Paths for Playing Media Objects. In *Proc. of the OpenArch'2001*, March 2001.
10. P. Reiher, R. Guy, M. Yavis, and A. Rudenko. Automated Planning for Open Architectures. In *Proc. of OpenArch'2000*, March 2000.
11. P. Sudame and B. Badrinath. Transformer Tunnels: A Framework for Providing Route-Specific Adaptations. In *Proc. of the USENIX Technical Conf.*, June 1998.
12. D. Tennenhouse and D. Wetherall. Towards an Active Network Architecture. *Computer Communications Review*, April 1996.
13. 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.
14. D. Xu and K. 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.
15. 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.