Available online at www.sciencedirect.com

SCIENCE DIRECT®

Computer Communications 28 (2005) 591–608

computer communications

www.elsevier.com/locate/comcom

Automatic creation and reconfiguration of network-aware service access paths

Xiaodong Fu*, Vijay Karamcheti

Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, New York, NY, USA

Received 5 August 2004; accepted 5 August 2004
Available online 11 September 2004

Abstract

This paper describes an adaptive network infrastructure, Composable Adaptive Network Services (CANS), for bridging the bandwidth and resource gap between network services and mobile clients. CANS enables construction of service access paths augmented with 'impedance matching' components that handle operations such as caching, protocol conversion, and content transcoding. The CANS infrastructure focuses on the *automatic creation* and *efficient dynamic reconfiguration* of such network-aware access paths, relying upon three key mechanisms: (a) a high-level integrated type-based specification of components and network resources; (b) an automatic path creation strategy; and (c) system support for low-overhead path reconfiguration.

We evaluate the CANS infrastructure over a range of network and end-device characteristics using two application scenario: web access and image streaming. Our results validate the effectiveness of the CANS approach for enabling network-aware service access to mobile clients, verifying that (1) communication paths automatically generated by CANS bring considerable performance advantages to applications; (2) desirable adaptation can be achieved using our flexible path creation mechanisms, which consider both underlying network conditions and different performance preferences of applications; and (3) despite their flexibility, both run-time overheads of CANS communication paths and reconfiguration time are negligible for most applications, providing applications with agile adaptation to dynamic changes in networks.
© 2004 Elsevier B.V. All rights reserved.

Keywords: Adaptive network infrastructure; Internet services; Quality of service (QoS); Adaptation

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 also experience very different connection characteristics over time, which may be caused by dynamic network load in shared network environments or mobility of the user. The user's interactions with the service should continually adapt to such changes; unfortunately, current infrastructures 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 ensuing this. Differentiated services, used in popular news and stock trading services, cannot satisfactorily handle users with connections exhibiting big variations in available bandwidth (e.g. a wireless LAN user at different distances from an access

* Corresponding author.
E-mail addresses: xiaodong@cs.nyu.edu (X. Fu), vijayk@cs.nyu.edu (V. Karamcheti).

point). The approach of including application specific adaptation logic into client and server applications, exemplified by automatic stream selection mechanisms in commercial media players, also has several limitations. First, encoding the adaptation into end applications make it hard to extend to cope with new problems. More importantly, the constraint that adaptation can only occur at end points may exclude good candidates for adaptation, and in many cases, such mechanisms may compromise adaptation agility to changes that occur in the middle of the network. For example, a media player that switches to a lower quality stream upon detecting congestion in the network middle could have avoided doing so if the stream was rerouted along a different path. Finally, building an effective adaptation solution using such explicit approaches requires considerable programming effort and comprehensive knowledge of network communication.

This paper describes a different approach to address this problem. Our approach, embodied in the Composable Adaptive Network Services (CANS) infrastructure, permits the dynamic insertion of application-specific components along the network path between the service and the client application. These components, which can transparently handle stream degradation, reconnection, and path rerouting in our example, and in general support arbitrary transcoding, caching, and protocol conversion operations, serve to 'impedance match' a user's performance requirements with underlying network conditions making it *network-aware*. CANS supports flexible mapping of these components to the hosts along a communication path. This flexibility permits CANS to uniformly cope with both diverse network conditions as well as changing load on shared resources. Although most mobile user scenarios are likely to benefit from components deployed in the last one or two networks hops, CANS can also be used in wide-area overlay networks to achieve increased control over the entire network path.

Other researchers have recently proposed similar programmable network infrastructures [1,5,8,12,24,27], however, CANS distinguishes itself by striving to create and dynamically reconfigure network-aware paths completely *automatically*. CANS achieves this goal using three mechanisms:

- A high-level integrated *type-based specification of components and network resources*, which enables late binding of components to paths, essential for flexibility of dynamic compositions.
- An *automatic path creation strategy* for constructing custom network-aware access paths according to application's specific performance preferences and underlying network conditions.
- System support for *low-overhead dynamic path reconfiguration*, providing applications with semantic continuity on data transmissions.

We have developed a prototype Java-based implementation of the CANS infrastructure, which is used in this paper to evaluate our approach, both in terms of the capabilities and performance of the constructed paths as well as the overheads for runtime component management, and communication path reconfiguration. We report on a series of experiments using two representative applications, web access and image streaming, in environments with different network and end-device characteristics. Our results validate the CANS approach, verifying that (1) automatic path creation and reconfiguration are achievable and do in fact yield substantial performance benefits; (2) our approach is effective at providing applications with fine tuned, desirable adaptation behaviors; (3) despite the flexibility, the overhead incurred by CANS infrastructure is negligible, and the cost to reconfigure communication paths is acceptable for most applications, which can be further substantially reduced by employing local planning and reconfiguration mechanisms.

The rest of this paper is organized as follows. Section 2 presents the overall CANS architecture. Sections 3–5 focus on the three mechanisms that enable automatic creation and efficient reconfiguration of network-aware paths, describing in turn the type framework, path creation strategy, and system support for path reconfiguration. Section 6 evaluates these mechanisms using the two applications. We discuss related work in Section 7 and conclude in Section 8.

2. CANS architecture

CANS is an application-level infrastructure for injecting application-specific components into the communication path between a client and a service. Traditionally, the functionality of a communication path has been restricted to transmitting data between the end points. The CANS infrastructure extends this notion to include various application specific functionality, which makes the access network aware. Such functionality, organized in the form of components, customizes the communication path with respect to the characteristics of the underlying network resources as well as enable it to automatically adapt to dynamic changes in these characteristics (see Fig. 1).

The CANS network view consists of *applications*, stateful *services*, and *communication paths* between them built up from mobile soft-state objects called *drivers*. Drivers serve as the basic building block for constructing adaptation-capable, customized communication paths. Drivers are standalone *mobile* code modules adhering to a restricted interface to permit their efficient composition and dynamic low-overhead reconfiguration. Specifically,

1. Drivers consume and produce data using a standard *data port* interface, called a `DPort`. `DPorts` are associated with type information (see Section 3 for details).

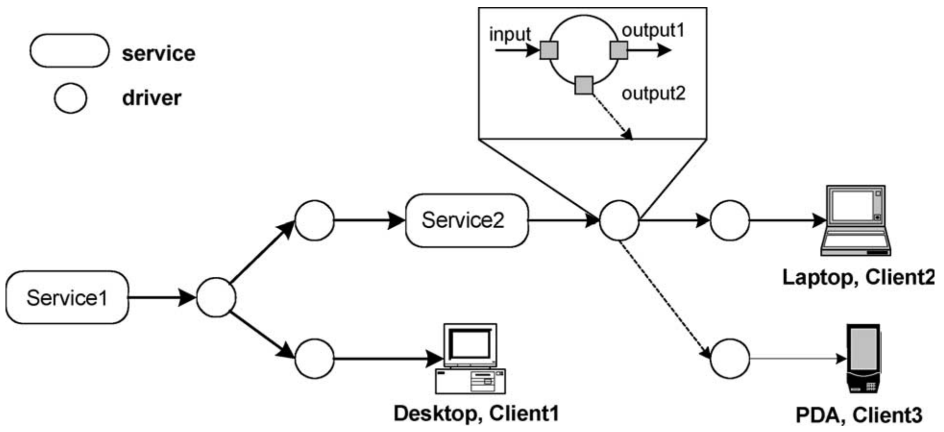


Fig. 1. Logical view of a CANS network showing data paths constructed from typed components.

2. Drivers are *passive*, moving data from input ports to output ports in a purely demand-driven fashion. Driver activity is triggered only when an output DPort is checked for data or an input DPort receives data.
3. Drivers consume and produce data at the granularity of an integral number of application-specific units, called *semantic segments*, e.g. an HTML page or an MPEG frame.
4. Drivers contain only *soft state*, which can be reconstructed simply by restarting the driver. Stated differently, given a semantically equivalent sequence of input segments, a soft-state driver always produces a semantically equivalent sequence of output segments.

The first two properties enable dynamic composition and efficient transfer of data segments between multiple drivers that are mapped to the same physical host (e.g. via shared memory). Moreover, they permit driver execution to be orchestrated for optimal performance. For example, a single thread can be employed to execute, in turn, multiple driver operations on a single data segment. The overhead between invocations of adjacent drivers is basically a few function calls, as if driver operations were statically combined into a single procedure call. The only additional overhead compared to using a statically linked module is the use of virtual functions. Finally, the passive operation greatly simplifies and enhances the efficiency of resource management among multiple paths by enabling control over resource consumption of individual paths within an execution environment. The semantic segments and soft-state properties enable low-overhead dynamic adaptation, a topic discussed in more detail in Section 5.2.

Services are the second core component. Unlike the constrained driver interface, services can export data using any standard protocol (e.g. TCP or HTTP), encapsulate heavyweight functions, process concurrent requests, and maintain persistent state. Relaxing interface requirements permits use of legacy services. However, CANS does not explicitly support service migration, requiring a service to manage its own state transfer.

The CANS network is realized by partitioning service and driver components belonging to multiple communication paths onto physical hosts, connected using existing communication mechanisms. Drivers run in CANS Execution Environments (EE) running on hosts along the network route.

To illustrate these concepts, let us consider the kinds of drivers that would enable the teleconference example we described earlier. The communication path for the mobile user should include a driver to handle network handoff. In addition, drivers that can reduce bandwidth requirements, for example by degrading the video quality, can be automatically inserted into the communication path in case of heavy cross traffic in the network. Furthermore, if the path crosses an unsecured link (e.g. a shared wireless network), encryption/decryption drivers should also be injected to protected sensitive data transmitted along the path.

Assumptions. We need to point out that there are two assumptions in the current version of the CANS infrastructure. First, we assume, there exists some support for distributed authentication and trust management among different network domains. Existing mechanisms, such as those used in PolicyMaker [4], KeyNote [3], Taos [25], and dRBAC [10], etc. can be integrated to allow users to express distributed trust relationships, and control downloading foreign code in CANS. Second, we assume that resource-monitoring functionality is available via some external entities. Mechanisms such as Refs. [6,18,19] can be incorporated in CANS, an issue we intend exploring in the near future.

3. Type-based component and resource specification

In CANS, the composability of components is determined by compatibility of type information of the input and output ports being connected. The basic idea is the notion that all data flowing along a data path is *typed*, and that this type is affected both by components along the communication path as well as network resources making up the route.

3.1. Representing component properties

CANS types integrate two concepts: *data types* and *stream types*. A related notion, *data type ranks*, helps capture application-specific composition constraints.

Data types are the basic unit of type information, represented by an object that in addition to a unique name can contain arbitrary attributes and operations for checking type compatibility. The CANS infrastructure assumes that in most application domains, it is possible to define a *closed*, semantically unambiguous set of types, e.g. MIME types to represent common media objects.

Traditional type hierarchies can still be used to organize data types; however, our scheme permits flexible type compatibility relationships not easily expressed just by matching type names. For instance, it is possible to define a customized `MPEG` type, which contains a frame size attribute such that it is compatible with any `MPEG` types with smaller frame size, naturally capturing the behavior that a lower resolution MPEG stream can be played on a client platform capable of displaying a higher resolution stream.

Stream types capture the aggregate effect of multiple drivers operating upon a data stream. Stream types are constructed at run time, and represented as a *stack* of data types. For example, after an MPEG type passes through an encryption driver, the stream type of its output port is a stack in which the type `Encryption` is placed on top of the type `MPEG` (Fig. 2).

The primary reason for using stream types is for eliminating the requirement for complete knowledge of the whole path when path segments need to be adjusted independently. By using stream types, any segment of a path only needs to consult its incoming and outgoing stream type instances.

This point is highlighted in Fig. 2 in which an MPEG type passes through an `Encryption` driver and a `Decryption` driver. If components were just modeled as consuming data of a particular type and producing data of another, if would be difficult to express the behavior of the `Encryption` and `Decryption` drivers in a way that permits their use with generic types *without* losing information about the original type at the output of the `Decryption` driver. Specifically, without stream types, the `Encryption` driver will set its output as being of the `Encrypted` type, and the output of the `Decryption` driver ends up being of the `BaseStream` type (unless the entire communication path is examined). This will cause a type compatibility problem at some downstream point because the client requires a more specific type (`MPEG`) than the incoming type (`BaseStream`). In contrast, the stream type representation preserves information about the specific type and thereby permits local decision making, which is important for run-time adaptation via dynamic component composition, especially for the cases where long communication paths are used.

Operations allowed on stream types included standard *push*, *peek*, and *clone*. From the type point of view, each CANS component with m input ports and n output ports defines a function that maps its input stream types into output stream types: $f(T_{\mathrm{in}_1}, \ldots, T_{\mathrm{in}_m}) \to (T_{\mathrm{out}_1}, \ldots, T_{\mathrm{out}_n})$ where T_{in_i} is the required data type set for the ith input port, and T_{out_j} is the resulting stream type produced on the jth output port. The type compatibility between an input and an output port is determined by checking the top of the output port's stream type against the required data type of the input port. Stream type information flows downstream automatically when two ports get connected at run time.

Data type ranks helps express application-specific constraints on the order of composition by requiring that only types of monotonically increasing ranks can be stacked into a stream type. For instance, giving the encryption type a higher rank, ensures that for any communication path requiring both encryption and compression, that encryption always happens after compression.

To simplify use of CANS in typical usage scenarios, our infrastructure predefines certain common data types encoding common operations such as encryption, compression, image transcoding, etc. These types are currently organized into a linear rank lattice. When a new type is added in the lattice, its constraints on type ranks will also be automatically checked by the system. Constraints on composition order, of course, can also be expressed using some rule-based mechanisms; however, our scheme is simple to use with our type model and quite expressive in describing various composition constraints. By using the notion of type ranks, valid composition patterns can be identified by only checking type compatibility between adjacent components and the type stacks that appear along the communication path.

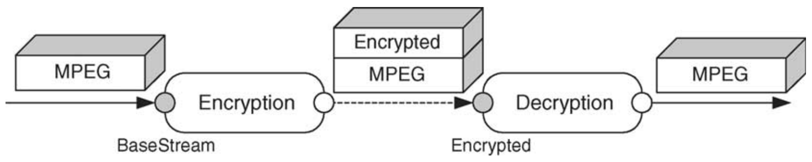


Fig. 2. An example of stream types.

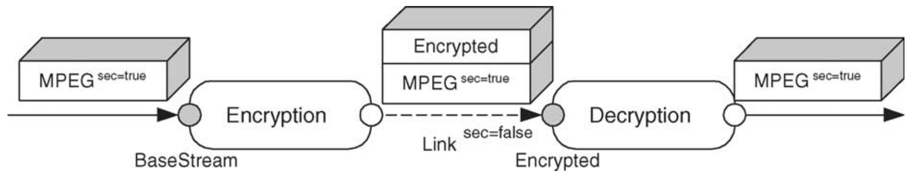


Fig. 3. A simple example of augmented types.

3.2. Representing network resource properties

Network resource characteristics can introduce additional constraints affecting both which components must be present along a communication path and how these can be composed. For example, the risk of packet interception on a shared wireless link necessitates the presence of encryption and decryption drivers to preserve privacy for data transmission. Since these drivers are not required if one just examines the type properties of the communication path source and sink locations, it is clear that one needs to factor in network resource characteristics into the component selection process. Unfortunately, prior research has usually modeled these resources in an ad hoc fashion, inserting components necessitated by characteristics such as link properties as a separate pass. While this approach works, it compromises optimality because of poor or redundant placement of these components.

In contrast, our approach unifies both type compatibility and network resource characteristics in the same framework. The basic idea of our approach is to represent network resource requirements implicitly by modeling how network resources affect the types of data that go across them.

To capture the effect of network resource properties on data types, we introduce the notion of *augmented types*: each data type is extended with a set of network resource properties that can take values from a fixed set such as security (used here to denote transmission privacy), reliability, and timeliness, etc. Network resources are modeled in terms of the same property set and have the effect of modifying, in a type-specific fashion, values of the corresponding properties associated with different data types. To consider an example, consider transmission of MPEG data over an insecure link. Our type framework captures this as follows: the data type produced at the source is represented by `MPEG`(*secure* = `true`), the network link is represented by the property *secure* = `false`, and the effect of the link property *secure* on the `MPEG` data type by the type-specific rule (i.e. defined by the `MPEG` type) that the augmented type `MPEG`(*secure* = `true`) is modified to `MPEG`(*secure* = `false`) upon crossing a link with the property *secure* = `false`.

This base scheme is extended to stream types by introducing the notion of *isolation*. Stated informally, specific data types have the capability of isolating others below them in the stream's type stack from having their properties be affected by network resources. For example, an `Encrypted` type can isolate the *secure* property of types that it 'wraps', i.e. this type of encrypted data still remains secure after crossing insecure links, irrespective of what specific type(s) the data corresponds to. Fig. 3 shows an example where an `MPEG` type crosses an insecure link.

With type compatibility to determine what component sequences are *valid*, the next step is to choose one of them and map it to underlying network resources to optimize desired performance metrics.

4. Automatic path creation strategy

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

The CANS path creation strategy automatically selects and maps 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 optimizes some overall path metric such as response time, data quality, or throughput. 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 will 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 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. After that, we discuss a local planning mechanism, which allows disjointed segments of a communication path to change their behavior independently and concurrently while maintaining some performance guarantee for the overall path. Lastly, we describe a distributed implementation of this strategy.

4.1. Base algorithm

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

A driver c is modeled in terms of its *computation load factor*(`load`(c)), the average per-input byte cost of running the component, and its *bandwidth impact factor* (`bwf`(c)), the average ratio between input and output data volume. We have found this simple model to be a reasonable approximation of the behavior of components in our experiments. We discuss incorporation of a more refined component model at the end of this subsection.

A *communication path*, $D=\{c_1,\ldots c_n\}$, is a sequence of type-compatible components. Type compatibility is defined in a *type graph* (G_t): vertices in the graph represent types, and edges represent components that can transform form the source type into the sink type.

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*, `comp`(n_i), and a link between two adjacent node n_j and n_{j+1}, denoted by l_j, is modeled in terms of its bandwidth, `bw`(l_j). Both `comp`(n_i) and `bw`(l_j) are defined in terms of route resources available for a particular path.

A mapping, $M{:}D\rightarrow R$, associates components on communication 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.

The component selection process takes as its input a route R, a source data type t_s, a destination data type t_d, and attempts to find a communication path D that transforms t_s to t_d and can be mapped to R to yield optimal value of some performance metrics, e.g. maximum throughput or minimal latency.

The problem as stated above is NP-hard. To make the problem tractable, we take a simplified 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 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.

4.1.1. 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 $\vec{A}$, consider all possible intermediate type 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', consider all possible mappings of the associated component c on nodes along the route that use no more than $\vec{A}$ resources. For each such mapping that transforms the available resource vector to $\vec{A}'$ (after accounting for `load`(c)), combine this component with the previously calculated solution for t' with i (or fewer) components with resource vector $\vec{A}$. The combined mapping that yields the maximum throughput is deemed the solution.

Because this procedure runs backwards from the sink to the source (i.e. c_{j+1} is mapped before c_j), consequently, only resource vectors of the form $(1,\ldots,1,\ r_j\in[0,1],\ 0,\ldots,0)$ 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 ($s[t_s, t, \vec{A}, i]$) 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 or fewer components and requiring no more resources that $\vec{A}$ ($\vec{A}\in RA$). Fig. 4 shows the moment during the calculation of $s[t_s, t_0, (1,1,3/4,0), i+1]$ when the component c is mapped to node n_3, and appended with previously calculated 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 four intervals.

The algorithm is shown in Fig. 5. Line 3 of the algorithm handles the base case: only the case $t=t_s$ achieves non-zero throughput. Lines 8–14 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 12–14 ensure that the component achieving the maximum throughput defines the next-level partial solution.

The algorithm terminates at Step $p\times n$, with the solution in $s[t_s, t_d, (1,\ldots,1), p\times n]$. This follows from the observation

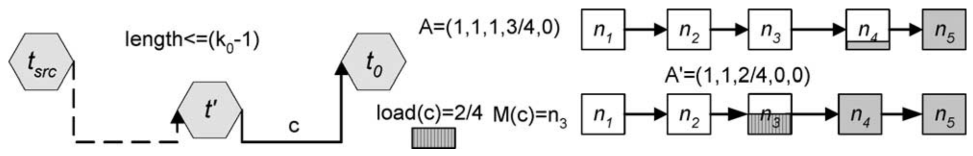


Fig. 4. Map c to n_3 and lookup solution with $\vec{A}'$.

```
Algorithm Plan
Input: $t_s$,$t_d$, $G_t$, $R$
Output: The communication 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 $c = (t', t) \in E(G_t)$
9.                     do for all $n_j$ that $\vec{A}[n_j] > 0$
10.                           do $M(c) \leftarrow n_j$
11.                              $\vec{A}' \leftarrow (\vec{A}[0], \ldots, \vec{A}[n_j - 1], \vec{A}[n_j] - \mathrm{load}(c), 0, \ldots)$
12.                              TH $\leftarrow$ throughput(append($s[t_s, t', \vec{A}', i-1], c, \vec{A}$))
13.                              if TH $> s[t_s, t, \vec{A}, i]$
14.                                then $s[t_s, t, \vec{A}, i] \leftarrow$ TH
15. return $s[t_s, t_d, \vec{A} = [1, 1, ..., 1], p \times n]$
```

Fig. 5. Base path creation algorithm.

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 exist many different candidates, not to mention that each component may have multiple configurations. Therefore, $O(p^n)$ 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.

4.1.2. Refining the component model

The simple component model described earlier can be extended in the following two ways. First, instead of supplying a single configuration (`load` and `bwf` pair), components are modeled ad possessing multiple configurations, which can be determined when a path is created or modified. Second, instead of being constant, the computation cost and compression ratio can be viewed as a function of attributes of the actual incoming stream type. 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 `Filter` component. The values of these parameters 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. Our experiment results in Section 6 show this refined model is important for obtaining desirable adaptation behaviors for applications.

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

4.2. Extension 1: planning for value ranges

Given that our planning algorithm constructs communication 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 solutions, 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 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 $s[t_s, t, \vec{A}, i]$, those partial solutions that cannot 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.

4.3. Extension 2: planning for path segments

Using 'local' schemes that can replace small portions of an existing communication path, disjoint segments of a communication path can adjust their behaviors independently and concurrently. Such support can considerably improve adaptation agility because a small portion of a communication path can be modified to respond to local changes in the network. More importantly, this is indispensable for deploying path-based infrastructures in situations where a communication path spans multiple network domains.

The challenge here is doing so while still being able to maintain some performance guarantee for the overall path: for example, that range metrics will still fall within their desired range. Note that local planning may compromise on optimality of performance metrics, but we look it as a reasonable tradeoff between the strict optimality of communication paths and responsiveness and scalability of such path-based infrastructures. We believe the latter is equally, not more important for CANS-like infrastructures to be used in networks of large scale.

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 (either statically or as observed at run time), the only thing left is to adjust the range metrics for R'. Adjustments for throughput and latency are shown below:

- For application that require a throughput range [th_{low}, th_{high}] for the overall path, this can be achieved by ensuring that each disjoint region in the path plans with the same range, which also gives them the most flexibility for building paths.
- For applications that require a latency range [l_{low}, l_{high}], the localized latency range is set to [$l_{(\text{low},R')}$, $l_{(\text{high},R')}$], where $l_{(V,R')}$ is the divided portion of latency l_V over segment R'. One way of doing this division is to consider the contribution of links in R' to overall latency or R.

4.4. Distributed (incremental) planning

Through 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

$$s\left[t_s, t, \vec{A} = (\underbrace{1,\ldots,1}_{i}, 0, \ldots, 0), \sum_{j=0}^{i}(CN_j)\right]$$

where CN_j is the total number of components in node n_j), and send these partial solutions to the next mode. 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 needs a complete knowledge of components and types for all nodes in the route. Instead, only knowledge of common types that are used across different network domains is required. This distributed version, combined with the local mechanisms described earlier, is important for a path-based system to be used in a wide area network, where a communication path usually spans multiple administration domains.

The traffic incurred for the distributed planning is just messages of partial solutions between adjacent nodes. It should be noted here that only values of the performance metrics needs to be present in the messages, transmission of the components themselves is unnecessary.

5. System support for efficient path reconfiguration

Network-aware communication paths need to be reconfigured to cope with dynamic changes in available resources. Our approach relies on two kinds of system support to enable low-overhead reconfiguration: (1) appropriate restrictions on component interfaces and (2) reconfiguration protocols that leverage these restrictions.

5.1. Reconfiguration semantics

The central question about reconfiguration is what the application can assume about transmitted data after a portion of the network path is reconfigured. CANS reconfiguration protocols can be customized to provide three levels of semantics:

- *Level 1* semantics provides no guarantees, leaving it up to the application to reconstruct any lost data.
- *Level 2* semantics provides the guarantee of delivering complete *semantic segments*, essentially simplifying the task of the application recovery code.
- *Level 3* semantics provide full continuity guarantees with exactly once semantics, completely isolating the application from the fact that the path has been reconfigured.[2]

5.2. Restrictions on the driver interface

To guarantee the above semantics, CANS relies upon the *semantic segment* and *soft state* properties of drivers, introduced in Section 2.

Semantic segments refer to demarcatable application-specific units of data transmission, e.g. an HTML page or an MPEG frame. CANS drivers are required to consume and produce data at the granularity of an integral number of semantic segments. Informally, this requirement ensures that the data in an input semantic segment can only influence data in a fixed number of output segments, permitting construction of communication path reconfiguration and error recovery strategies that rely upon retransmission at the granularity of semantic segments.

[2] Note that real-time applications can still detect a break in data availability; we take the view that such applications are best handled by inserting additional application-specific components that provide necessary timeliness guarantees.

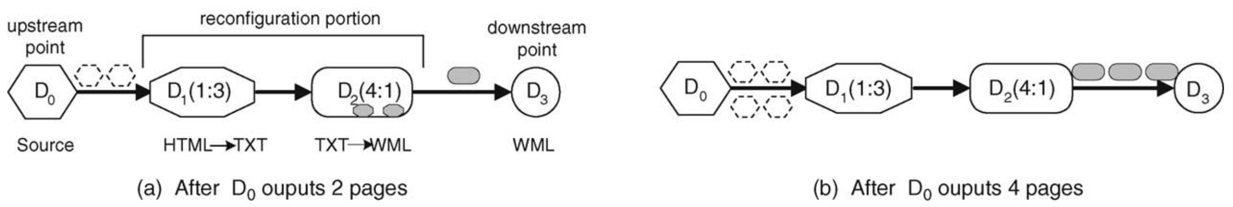


Fig. 6. An example of communication path reconfiguration using semantics segments.

Soft state refers to the driver property, which allows internal state to be reconstructed simply by restarting the driver. Stated differently, given a semantically equivalent sequence of input segments, a soft-state driver always produces a semantically equivalent sequence of output segments.

Together, these two properties enable low-overhead path reconfiguration as described in Section 5.3.

5.3. Reconfiguration protocol

The reconfiguration process is triggered by dynamic changes, and consists of three major steps:[3] (1) generation of a new plan; (2) ensuring required semantics prior to suspending data transmission; and (3) deploying the new plan and resuming data transmission. The primary problem is that to maintain semantic continuity and exactly once semantics,[4] any scheme must take into account the fact that the portion of the communication path being reconfigured can have stream data that may be buffered in the internal state of drivers, or in transit between execution environments.

Fig. 6 shows an example highlighting this problem. To introduce some terminology, we refer to the portion of a communication path that needs to be modified due to changes in the network as the *reconfiguration portion*, and the components immediately upstream and downstream of this portion as the *upstream point* and *downstream point*, respectively. In the example, driver d_0 is an HTML data source, and d_3 is a component receiving WML data. The reconfiguration portion consists of drivers d_1 and d_2. In this case, let's assume that driver d_1 converts every incoming HTML page into three TXT pages, and driver d_2 composes every four incoming TXT pages into a WML deck. Consider a situation where system conditions change after the upstream point d_0 has output two HTML pages, and the downstream point d_3 has received one WML deck. At this point, the reconfiguration portion cannot be replaced because doing so affects semantic continuity. It is incorrect to retransmit either the second page from d_0 whose effects have been partially observed at d_3, or the third page, which would result in a loss of continuity at d_3.

The basic idea of our solution is to delay the reconfiguration to *safe points* in data transmission where the reconfiguration portion can be safely removed, and semantic continuity can be achieved using *selective retransmission* of data that has not been seen downstream of the reconfiguration portion.

The key to detecting these 'safe' points is keep track of the correspondence between segments received at the downstream point and the segments sent from the upstream point, which is determined by the driver characteristics in the reconfiguration portion. If a reconfiguration portion contains a sequence of drivers $D=\{c_1,\ldots,c_n\}$ of which driver c_i produces p_i semantic segments upon receiving q_i input segments, we refer to $p/q=\prod_{i=1}^{i=n} p_i/q_i$ as the *synthesis factor* of the reconfiguration portion (here p and q are relative primes). For the reconfiguration portion, the semantic information in the jth outgoing segment from the upstream point is contained in segments within the range of $[\lfloor j\times p/q\rfloor, \lfloor j\times p/q\rfloor]$ received by the downstream point. More interesting is the fact that the boundary of each $(i\times q)$th segment at the upstream point is preserved at the downstream point, which corresponds to the boundary of the $(i\times p)$th segment. This means that after the downstream point receives such a segment, all segments (inclusively) before the $(i\times q)$th segment must have been seen at the downstream point and there is no state of these segments left in the reconfiguration portion. Such segments are referred to as *flushing* segments in our reconfiguration protocol to reflect the fact that these segments can in effect completely push state (and data) remaining in the reconfiguration portion (of previous segments) to the downstream point.

Note that in practice p_i/q_i may not necessarily be a constant number, so our framework exploits a flexible mechanism that tracks these flushing segments by using marker messages, which demarcate segment boundaries. All drivers along a communication path are required to pass only incoming markers that match their output segment boundary (others will be discarded). Therefore, receipt of a marker at the downstream point of a configuration portion signifies the end of a flushing segment (sent by the upstream point).

5.4. Reconfiguration process

The state diagram of the path during reconfiguration is depicted in Fig. 7. In this figure, bold font is used for distinguishing control messages or data segments from

[3] These steps may be overlapped with each other.

[4] Since activities for Levels 1 and 2 are a subset of that for Level 3, our description focuses on the latter.

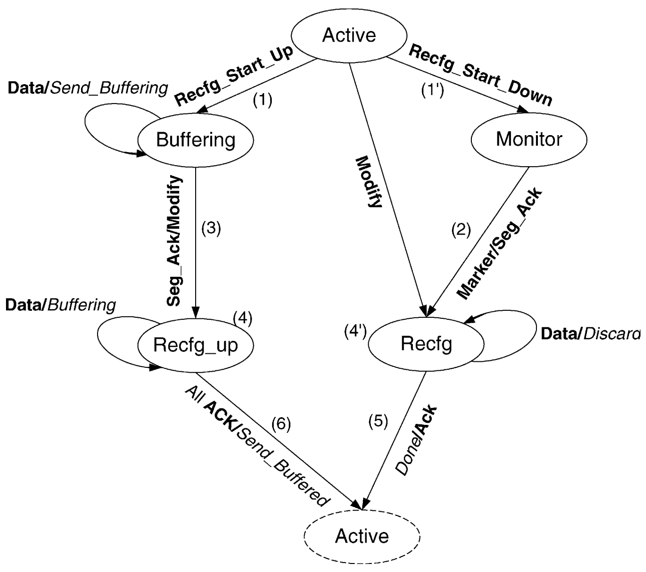


Fig. 7. State diagram of path reconfiguration. Numbers on arcs correspond to the steps described in the text.

actions taken by the path (shown in italics). The reconfiguration process includes the following steps.

1. Upon receiving a message signifying the start of a reconfiguration (with the new plan), the downstream point starts to monitor incoming data ($1'$) (the `Monitor` state); the upstream point starts to buffer outgoing segments while continuing to deliver them downstream (1) (the `Buffering` state). Besides, a marker is appended at the end of each output segment from the upstream point. Other nodes within the reconfiguration portion do not change their state.
2. The downstream point continues monitoring until it receives a marker from the upstream point, which signifies the end of a flushing segment from the upstream point. The downstream point then sends a *Seg_Ack* message to the upstream point, and begins to discard any further incoming data segments (2).
3. Upon receipt of the Seg_Ack from the downstream point, the upstream point suspends (keeps buffering but does not deliver data downstream) data transmission and sends a *Modify* message to all nodes that are involved in the reconfiguration (3).
4. Upon receipt of a *Modify* message, all nodes in the reconfiguration portion enters the `Recfg` state, tearing down the components in the old configuration and replacing them with the new component graph. In this stage, all drivers within the reconfiguration portion except the upstream point discard any incoming data ($4'$). The upstream point continues buffering outgoing segments (4).
5. After the modification on a node is finished, an *ACK* messages will be sent to the upstream point (5).
6. After receiving ACK messages from all nodes, the upstream point resumes data transmission, starting with retransmission from the segment that follows the last flushing segment received by the downstream point (6). Note that every driver is associated with a unique ID, so the new components will not receive the data in the network that is addressed to the old ones.

The process described above achieves Level 3 reconfiguration semantics. For semantics Level 2, the data buffering and retransmission actions can be omitted. For semantics Level 1, step 1 and 2 can be further bypassed, i.e. the upstream point need not wait for the Seg_Ack message from the downstream point.

Example. For the example shown in Fig. 6, reconfiguration works as follows. First, the upstream point (d_0) starts buffering every segment it produces after the reconfiguration beings. The downstream point (d_3) will receive a marker after the third page form d_2, which is the marker appended at the end of the fourth page from the upstream point. It then sends an acknowledgement to the upstream point. After that, data transmission will be suspended at d_0 so that d_1 and d_2 can be replaced with another compatible driver graph. To resume data transmission, d_0 retransmits buffered data starting from the fifth page.

5.5. Error recovery

In addition to adapting to changes in resource availability, our scheme can also be used for 'extreme' cases where link or node errors cause loss of data or driver state. The only difference between the two situations is whether the reconfiguration protocol is executed on demand or runs all the time.

To gracefully recover from the failures of links and nodes along a communication path, we need to do buffering and monitoring all the time at the upstream and downstream points of an unstable network segment. Moreover, the downstream point needs to delay the delivery of received segments until it receives a marker from the upstream point. Meanwhile, the downstream also needs to send acknowledgements of received markers to the upstream point so that the upstream point can free buffer space accordingly. Upon recovery from a network failure, the downstream point discards its buffered data and resends the acknowledgement of the last received marker to the upstream point. The procedure that follows is exactly the same as steps (3)–(6) in the reconfiguration process described earlier.

5.6. Local reconfiguration

In addition to modifying a whole communication path (*global reconfiguration*), the reconfiguration process can also be applied to allow individual nodes or small portions of a communication path to adjust their behaviors

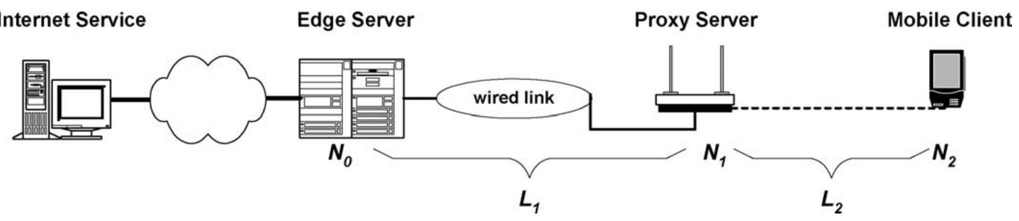


Fig. 8. A typical network path between a mobile client and an internet services.

independently and concurrently. We refer to the latter as *local reconfiguration*. By using local reconfiguration, every segment of a communication path can independently and concurrently adapt to dynamic changes in the network. This not only results in better responsiveness, but more importantly, also enables each network domain along the path to control its portion independently, and is especially important for infrastructures that run in a wide are network where fine-grained coordination across different network domains is either prohibitively expensive or infeasible due to administration policies.

To support local reconfiguration, in addition to the reconfiguration process described earlier, we need two more things. First, we need a planning algorithm suitable for generating a small path portion to replace a part of an existing communication path while retaining some overall performance guarantee. In Section 4.3, we have described an algorithm that can provide such support. Second, we need strategies to determine which part of a communication path (i.e. nodes and links) should be involved in a local reconfiguration. In this section, we focus on this issue.

To start with an example, if the bandwidth of a network link changes, local reconfiguration may first try replanning only the direct upstream node of that link. If the new calculated plan can cope with the change, it will be deployed without further action. Otherwise, the reconfiguration portion has to be increased to involve more network resources until the situation is handled. Note that this propagation can be terminated at any time by just invoking a global reconfiguration.

The tradeoff in choosing an appropriate point to switch between local and global reconfiguration involves the length of the segment selected for reconfiguration (which affects reconfiguration cost), and the likelihood that the reconfiguration can successfully handle changes. Our framework uses a three-level strategy. Upon a reconfiguration request, the first reconfiguration attempt happens at a single node whenever its load changes or the load on its direct downstream link changes. If the first reconfiguration attempt cannot meet the application requirements, then the second reconfiguration attempt will be triggered, which includes network segments comprising nodes connected with relatively fast links (usually within a single network domain). If both of these attempts fail, a global reconfiguration will be started for the whole communication path.

6. Performance evaluation

To evaluate the effectiveness of CANS mechanisms detailed in Sections 3–5 in enabling automatic creation and reconfiguration, we built a Java-based prototype of the CANS infrastructure 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, we measured the performance advantage brought by automatically generated paths, using the web access application, under a wide range of network conditions.

Second, we observed the continuous adaptation behaviors achieved with CANS, using the image application within a shared wireless environment, where available bandwidth changes frequently.

Last, we measured overhead of CANS mechanisms, including reconfiguration costs and the overhead incurred by the CANS run-time system for managing components along communication paths.

6.1. Experimental platform

In this paper, we consider a typical network path between a mobile client and an Internet server as shown in Fig. 8. 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).[5]

The *web access application* is a browser client, which downloads web pages (both HTML page and images). For this application, short response time is the major performance concern.

[5] 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.

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

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

Relative computation power of different node types (normalized to a 1 GHz Pentium III node with 256 MB 800 MHz RDRAM): 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.

The *image streaming application* is a simple Java-based application that continuously fetches JPEG frames from an image server. To perform appropriately, this application requires that frames arrive at a certain throughput (i.e. frames/s) and prefers high quality data.

6.2. *Effectiveness of automatic path creation*

Components used with the web access application include: `ImageFilter` and `ImageResizer` components, which can reduce bandwidth usage for images by degrading image quality to a factor of 0.2 and reducing image size to a factor of 0.2, respectively, *Zip* and *Unzip* components, which work together to compress text pages. 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 experiments 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 evaluating whether our approach could effectively adapt to multiple network conditions. Evaluating the effectiveness of the approach when component characteristics may be imprecise is investigated in our second set of experiments.

To model different network conditions likely to be encountered along a mobile access path, we defined 12 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.[6] 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 using ‘sandboxing’ techniques that constrain CPU, memory, and network resources available to an application [7]. The computation power of different nodes is normalized to a 1 GHz Pentium III node with 256 MB, 800 MHz RDRAM.

Table 1 also identifies, for each platform configuration, the plan automatically generated by CANS for the web access application. The plans themselves are shown in Table 2, listing the components deployed along the Image and Text paths. To take an example, consider platform configuration 7 for which the path creation strategy generates Plan C. The reason for this plan is as follows. Since link L_1 has high bandwidth while L_2 has moderate bandwidth, there is a need to reduce image transmission size, which is accomplished using the `ImageFilter` component. The `Zip` and `Unzip` drivers help improve download speeds by trading off computation for network bandwidth. Both the `ImageFilter` and `Zip` components are placed on the proxy server, because it has more capacity than the edge server.

Fig. 9 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 Fig. 9 are normalized with respect to the best response time achieved on each platform (so lower is better). In all 12 configurations, the generated plans improve the response time metric, by up to a factor of seven. Note that the lower response times come at the cost of degraded

Table 2
Component placement for the five automatically generated plans

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

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

image quality, but this is to be expected. The point here is that our approach *automates* the decisions of when such degradation is necessary. Fig. 9 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 CANS-generated plan is the one that yields the best performance, also improving performance by up to a factor of seven over the worst-performing transcoding path.

6.3. Adaptation behaviors

To investigate the kinds of continuous adaptation behaviors that can be achieved by CANS-like approaches, we run a set of experiments with the image streaming 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 s (modeling new user arrivals or movement away from the access point) before rising back 150 KBps at the same rate (modeling user departures or movement towards the access point). The communication 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 components used with this image streaming application included the `ImageFilter` and `ImageResizer` used before. To display incoming images appropriately, the throughput is required to be between 8 and 15 frames/s. Within that range, better image quality is preferred. We started with the base planning strategy described in Section 4.1. Since it can only optimize one of these metrics at a time, we chose to optimize throughput. The component parameters were obtained by profiling their behavior on a 25 KB JPEG image (quality assumed to be 1.0), one of a set of images repeatedly transmitted by the server ranging in size from 20 to 30 KB.

Fig. 10 shows the throughput and image quality achieved by the communication path over the 20 min run of the experiment; the plans automatically deployed by CANS are shown in the right table. 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/s, and a bandwidth of 10 KBps corresponds to a throughput of 0.4 frames/s. 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 labeled with the plan that is deployed during the corresponding time interval.

The results in Fig. 10 show that the plans automatically created and dynamically deployed by CANS 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 preference.

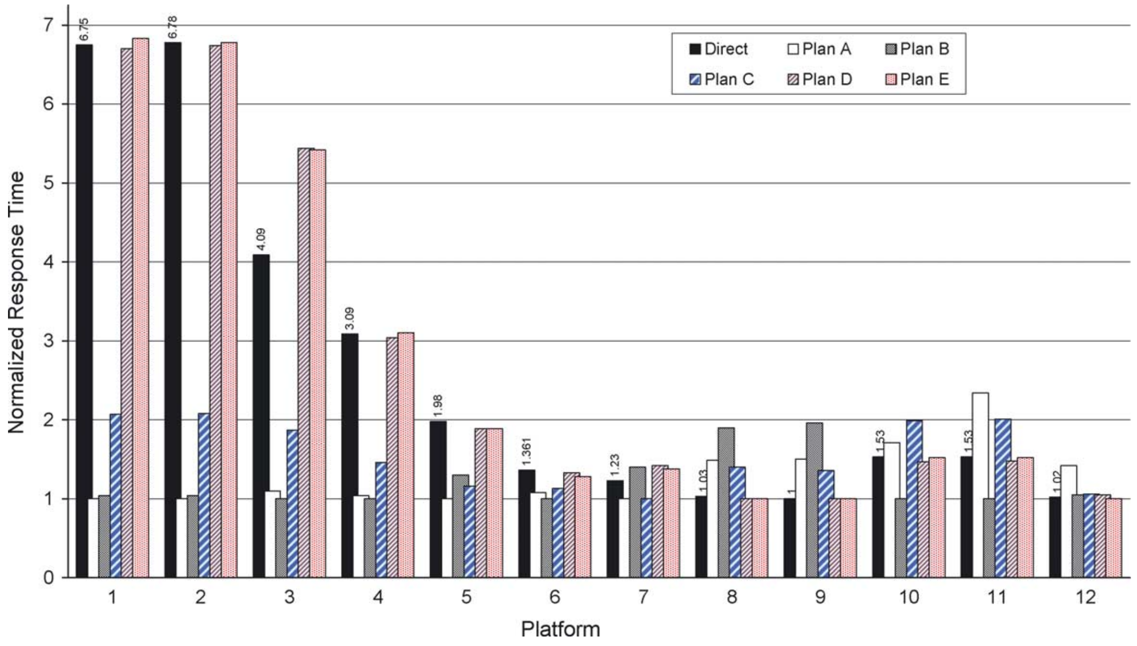
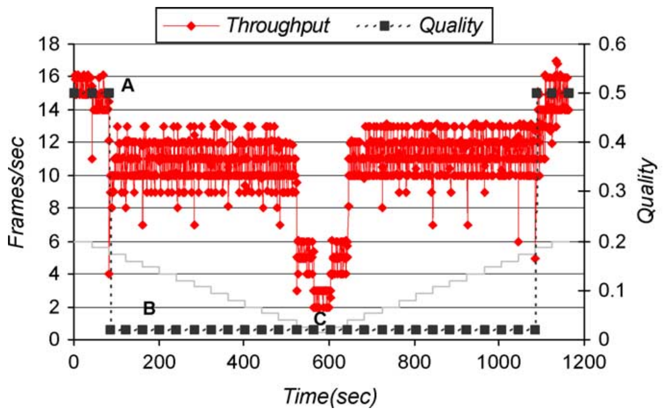


Fig. 9. Response times achieved by different plans for each of the 12 platform configurations compared to that achieved by direct interaction. All times are normalized to the best performing plan for each configuration.



Plan	N_0	N_1
A	**F**	–
B	**F**	**R**
C	**F R**	-

F: Filter, **R**: Resizer

Fig. 10. Performance of CANS base mechanisms.

- The reconfiguration at 80 s from Plan A to Plan B is seemingly unexplainable give that it was initiated to improve application throughput, not to reduce it. A closer examination identified this problem is caused by the fact that component behavior for the `ImageResizer` component did not match profiled behavior when the input was a filtered image as opposed to the original. Similar problem exists for Plan C.

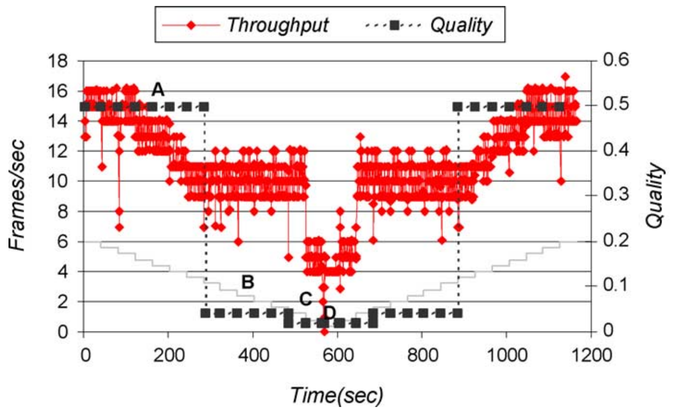
To address the first problem, we applied the range planning algorithm (Section 4.2) to this application, and obtained the results shown in Fig. 11. Comparing with Fig. 10, we can see two improvements. First, the range planning system retains Plan A for much longer than before (till 280 s into the experiment), 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 Fig. 10 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 desirable adaptation expected by end users.

To address undesirable adaptation caused by inaccurate component parameters, we exploited the refined component model described earlier. In particular, we allowed both components in our image streaming example to take on multiple configuration: nine `Filter` configurations corresponding to quality values 0.1–0.9, and four `Resizer` configurations corresponding to scale factors of 0.2, 0.4, 0.6, and 0.8. In additional, the components were profiled for three different image classes (high, medium, low) (*class profiling*). Fig. 12 shows the resulting performance and associated plans. There are three obvious improvements over Fig. 11. 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 what was previously shown in Fig. 11. Instead of three configurations (quality levels), there are seven different plans, permitting 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 (transition between Plan C and D).

6.4. Overhead of CANS mechanisms

6.4.1. Runtime overhead

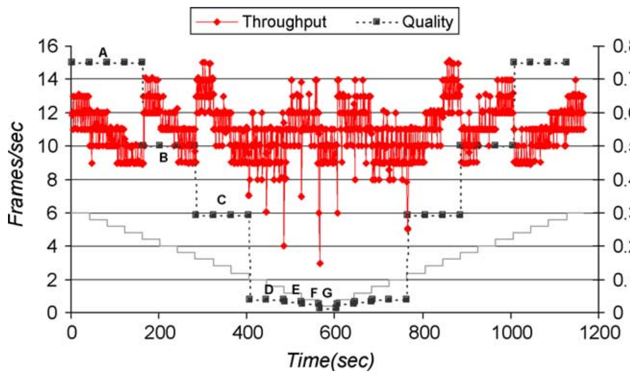
To understand the run-time overheads incurred by the CANS infrastructure, we first measured the impact on bandwidth and latency of communication paths when CANS execution environments are employed in the middle



Plan	N_0	N_1
A	**F**	-
B	**R**	-
C	**F**	**R**
D	**F R**	-

F: Filter, **R**: Resizer

Fig. 11. Performance with range planning support.



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

F: Filter, **R**: Resizer

Fig. 12. Performance with multi-configuration components and class profiling.

of the network. The results show that each intermediate CANS execution environment along a communication path introduces an extra 1.5 ms in the round trip time (i.e. 0.75 ms one way) for a 4K packet (on a Pentium II 450 MHz with 128 MB PC 100 SDRAM). For bandwidth, such paths can sustain about 70 Mbps where running in a 100 Mbps LAN environment. Given that intermediate EEs are intended to be used across different network domains in the Internet where other factors dominate latency and bandwidth, such overhead is unlikely to have much overall impact.

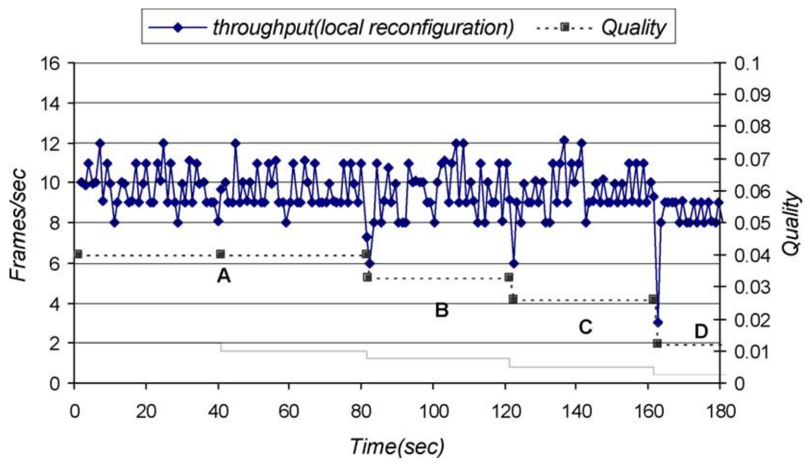
We also profiled both applications with our implementation to construct a detailed timeline of operations. CANS incurs an average cost of 25 μs per driver invocation, which is negligible for most data-processing components.

6.4.2. Reconfiguration overhead

We measured the cost of communication path reconfiguration using the image streaming application, with local and global reconfiguration, respectively. In both cases, we measure the cost for Level 3 reconfiguration.

To emphasize the difference in behaviors between local and global reconfiguration, we closely examined the portion of the experiment between 400 and 600 s (shown in Figs. 13 and 14), corresponding to a bandwidth range of 50–10 KBps. Unlike global reconfiguration which partitions the `ImageResizer` and `ImageFilter` portions of the communication 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 chooses to both calculate the plan and deploy the components on the same node, thereby avoiding the cost of coordination across nodes. The cost, however, is that the local reconfiguration does not quite achieve the same throughput as the global case, achieving 10 frames/s instead of 12. Note that this is still within the desired range, otherwise global reconfiguration would have been triggered.

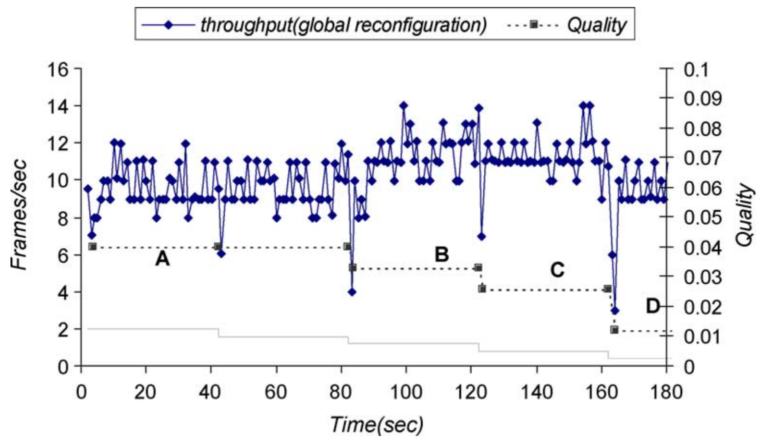
A breakdown of the reconfiguration costs for the bandwidth change event at 480 s in the two cases is shown in Fig. 15. The total reconfiguration time is 1.08 and 0.35 s for the global and local case, respectively. This figure shows that the major contributors to shorter reconfiguration times in the local mechanism are the first three stages of reconfiguration: shorter planning time, which is the result of shorter network paths; and shorter overheads for partitioning the plan, flushing data belonging to the old plan, and deploying the new plan, all of which benefit from the fact that all required coordination occurs locally and there is less data in transit. It also needs to be noted that during the first three stages of reconfiguration, data keeps flowing downstream.



Plan	N_0	N_1
A	-	**R**(1)
B	-	**R**(1),**F**(5)
C	-	**R**(1),**F**(3)
D	-	**R**(1),**F**(1)

F: Filter, **R**: Resizer

Fig. 13. Performance of local reconfiguration.



Plan	N_0	N_1
A	-	**R**(1)
B	**R**(1)	**F**(5)
C	**R**(1)	**F**(3)
D	**R**(1)	**F**(1)

F: Filter, **R**: Resizer

Fig. 14. Performance of global reconfiguration.

So the suspension period of data transmission is about 0.18 s for the global case and 0.12 s for the local case. The time for first three stages basically reflects the inertia of communication paths, in which the existing path is still in use after a new change is detected. The difference (about 0.68 s) between global and local mechanisms means that using local mechanisms can substantially increase the responsiveness of the communication path. From Figs. 13 and 14, we can observe that the use of local reconfigurations does result in more stable throughput during reconfiguration (look especially at the first reconfiguration that happens 480 s into the experiment).

6.5. Summary

The experiment result described in this section verify that (1) network-aware communication paths can be automatically created and dynamically configure; (2) these automatically created paths provide considerable performance advantage for applications; (3) our mechanisms can provide applications with fine tuned, desirable continuous adaptation behaviors; (4) the run-time overheads of CANS paths is negligible, and reconfiguration cost is small for most applications, and can be further reduced by our local mechanisms.

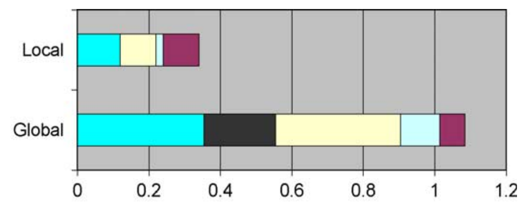
7. Discussion and related work

Our work is related to works on adaptation frameworks that can improve user experience by reacting to changes in network conditions. Though many application-specific approaches have been proposed and some of them have gained tremendous success, we believe general application-independent mechanisms are more attractive because of their wider applicability. Furthermore, using such approaches can also simplify the construction of approaches aiming at one specific application. General adaptation frameworks can be categorized into three groups: end-point approaches, proxy-based, and path based approaches.

End-point approaches such as Odyssey [22], Rover [16], and InfoPyramid [20] provide flexible platforms for applications to adapt to changes in network environments. However, limiting adaptation behaviors only at end points hampers the responsiveness for coping with changes at intermediate points in the network. Moreover, it is usually hard to deploy such approaches on small portable devices with strict resource constraints.

The cluster-based proxies in BARWAN/Daedalus [8], TACC [9], MultiSpace [13], and Active Services [1] are examples of systems where application-transparent adaptation happens in intermediate proxy nodes in the network. Compared with end-point approaches, proxy-based infrastructures provide resource sharing at proxy sites thus it can work with servers or client devices with limited computation capability. But on the other hand, the limitation that adaptation only happens at the last hop can cause considerable resource waste in the middle of the network.

Path-based mechanisms provide more flexible and responsive solutions. Our experiment results in Section 6 verify that adaptation occurring close to the actual change point does significantly reduce the response time. More importantly, injecting computation into the whole network not only enables such an approach to work with small server sites or weak clients, but also results in better performance of the overall network. Detailed information about



	Global	Local
■ 5. Firstst Message	0.07	0.1
□ 4. Reconfiguration	0.11	0.02
□ 3. Flush-Data	0.35	0.1
■ 2. Partition-Shipping	0.2	0
■ 1. Planning	0.354	0.12

Fig. 15. Reconfiguration cost.

performance differences among these approaches can be found in a study described in Ref. [11].

Among path-based infrastructures, the Ninja project's Automatic Path Creation (APC) service [12], also used by Kiciman and Fox in their mediator-based path framework [17], allows creation of paths between various end devices and services. Both APC and CANS formulate the component selection problem in terms of type compatibility, however, unlike CANS performance-oriented focus, APC is a function-oriented method, which ignores network resource properties and constraints, and thus is not able to provide applications with optimized performance. Other differences include our support for path reconfiguration and a more general notion of data, stream, and augmented types.

Template-based or reusable plan set are used in the Scout project [21] and the Panda project [23]. 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 [26] 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. The work [15] 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 path creation strategies solve 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.

Though these infrastructures can enhance application performance to some extent, the network awareness provided by them is limited in that they lack the mechanisms for constructing communication paths with optimized performance and dynamically modifying paths when network conditions change. Differing from these infrastructures, CANS focuses on providing network-aware communication paths that can continually cope with changes in *dynamic* network environments and provide application with optimized performance, requiring only minimum input from applications. We achieve this goal using (1) effective data path creation strategies that satisfy applications' performance needs, and provide desirable adaptation behavior, (2) low overhead communication path reconfiguration, enabling agile adaptation to dynamic changes, and (3) local and distributed schemes that make CANS applicable for wide area networks.

Limitations of the CANS Approach. 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. Three notable omissions in this work include (1) the issue of encoding component semantics in a high-level fashion (as opposed to just specifying its input and output data types); (2) dealing with security concerns for paths spanning multiple administrative domains (though our local and distributed schemes considerably reduce dependency between different network domains); and (3) integrating resource monitoring information. The first issue needs some clarification. In its current form, the CANS infrastructure is most suitable for deploying components that transform the format of data presentation, but still retain the same semantic information (e.g. a transcoded web page still represents the original). However, enabling use of semantics-transforming components would require going beyond simple types, possibly leveraging standard ontologies such as being developed by the Semantic Web [2] and IEEE's Standard Upper Ontology [14] efforts.

8. Conclusion

This paper has presented and evaluated an automatic approach for the dynamic deployment of intermediary components along client–server paths, which can be efficiently reconfigured at run time, to enable ubiquitous network-aware access to services. In contrast to current-day approaches relying on intermediate static proxies, our work argues for a flexible approach where the entire paths leading to these services are automatically and dynamically reconfigured to satisfy user preferences and network resource constraints. Using our approach, regular applications can easily be augmented with the capability of adapting to changes in the network. The experimental results show that such network-aware communication paths cannot only bring applications considerable performance benefits, but also provide desirable adaptation behaviors in dynamic environments. The local and distributed schemes built in the CANS infrastructure make it suitable for used in wide area networks.

References

[1] E. Amir, S. McCanne, R. Katz, An active service framework and its application to real-time multimedia transcoding, in: Proceedings of the SIGCOMM'98, August 1998.

[2] T.B. Lee, Services and semantics: web architecture, in: http://www.w3.org/2001/04/30-tbl.html, 2001.

[3] M. Blaze, J. Feigenbaum, A.D. Keromytis, KeyNote: trust management for public-key infrastructures (position paper), Lecture Notes in Computer Science 1550 (1999) 59–63.

[4] M. Blaze, J. Feigenbaum, J. Lacy, Decentralized trust management, in: Proceedings of IEEE Conference on Privacy and Security, 1996.

[5] A.T. Campbell, et al., A survey of programmable networks, ACM SIGCOMM Computer Communication Review 1999;.

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

[7] F. Chang, A. Itzkovitz, V. Karamcheti, User-level resource-constrained sandboxing, in: Proceedings of the Fourth USENIX Windows Systems Symposium, August 2000.

[8] A. Fox, S. Gribble, Y. Chawathe, E.A. Brewer, Adapting to network and client variation using infrastructural proxies: lessons and prespectives, IEEE Personal Communication 1998;.

[9] A. Fox, S. Gribble, Y. Chawathe, E.A. Brewer, P. Gauthier, Cluster-based scalable network services, in: Proceedings of the 16th ACM Symposium on Operating Systems Principles, October 1997.

[10] E. Freudenthal, T. Pesin, E. Keenan, L. Port, 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.

[11] X. Fu, V. Karamcheti, Why path based adaptation? performance implications of different adaptation mechanisms for network content delivery, Technical Report TR2003-843, New York University, July 2003.

[12] S.D. Gribble, et al., The ninja architecture for robust internet-scale systems and services, Special Issue of IEEE Computer Networks on Pervasive, 2000.

[13] S.D. Gribble, M. Welsh, E.A. Brewer, D. Culler, The MultiSpace: an evolutionary platform for infrastructual services, in: Proceedings of the 1999 Usenix Annual Technical Conference, June 1999.

[14] Standard Upper Ontology (SUO) Working Group. IEEE standard upper ontology scope and purpose, in: http://suo.ieee.org/scopeAndPurpose.html, 2001.

[15] X. Gu, K. Nahrstedt, R.N. Chang, C. Ward, Qos-assured service composition in managed service overlay networks, in: Proceedings of the 23rd International Conference on Distributed Computing Systems, May 2003.

[16] A.D. Joseph, J.A. Tauber, M.F. Kasshoek, Mobile computing with the rover toolkit, IEEE Transaction on Computers: Special Issue on Mobile Computing 46 (3) (1997).

[17] E. Kiciman, A. Fox, Using dynamic mediation to intergrate COTS entities in a ubiquitious computing environment, in: Proceedings of the Second Handheld and Ubiquitous Computing Conference (HUC'00), March 2000.

[18] K. Lai, M. Baker, Nettimer: a tool for measuring bottleneck link bandwidth, in: Proceedings of the Third USENIX Symposium on Internet Technologies and Systems (USITS), March 2001.

[19] B. Lowekamp, N. Miller, D. Sutherland, T. Gross, P. Steenkiste, J. Subhlok, A resource query interface for network-aware applications, in: Seventh IEEE Symposium on High-Performance Distributed Computing, July 1998.

[20] R. Mohan, J.R. Simth, C.S. Li, Adapting multimedia internet content for universal access, IEEE Transactions on Multimedia 1 (1) (1999) 104–114.

[21] A. Nakao, L. Peterson, A. Bavier, Constructing end-to-end paths for playing media objects, in: Proceedings of the OpenArch'2001, March, 2001

[22] B.D. Noble, Mobile data access, PhD Thesis, School of Computer Science, Carnegie Mellon University, 1998.

[23] P. Reiher, R. Guy, M. Yavis, A. Rudenko, Automated planning for open architectures, in: Proceedings of OpenArch'2000, March. 2000.

[24] D. Tennenhouse, D. Wetherall, Towards an active network architecture, Computer Communications Review 1996;.

[25] E. Wobber, M. Abadi, M. Burrows, B. Lampson, Authentication in the taos operation system, ACM Transactions on Computer Systems 1994; 3–32.

[26] D. Xu, K. Nahrstedt, Finding service paths in a media service proxy network, in: Proceedings of SPIE/ACM Conference on Multimedia Computing and Networking (MMCN 2002), Jan 2002.

[27] M. Yavis, A. Wang, A. Rudenko, P. Reiher, G.J. Popek, Conductor: distributed adaptation for complex networks, in: Proceedings of the Seventh Workshop on Hot Topics in Operating Systems, March 1999.