Automatic Network-Aware Service Access

Xiaodong Fu, Weisong Shi, Vijay Karamcheti
Department of Computer Science
Courant Institute of Mathematical Sciences
New York University

Abstract

Although advances in wireless networks and the increasing availability of mobile end devices raise the prospect of ubiquitous access to network-based services, such access must cope with an inherent mismatch between the high bandwidth, relatively static connection requirements of many services and the low bandwidth, limited resource, and dynamic nature of mobile clients.

In this paper, we describe an application-level programmable network infrastructure called CANS (Composable Adaptive Network Services), which alleviates this mismatch by enabling 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 *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 scenarios, web access and image streaming, with client preferences for reduced response time and increased throughput respectively. Our results validate the effectiveness of the CANS approach for enabling network-aware service access to mobile clients, verifying that (1) automatic path creation and reconfiguration is achievable and desirable; and (2) that despite their flexibility, both path creation and reconfiguration can be supported with low run-time overhead.

1 Introduction

Advances in wireless networking, combined with the growing number of communication-enabled portable end-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 an example 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. Chief amongst these is coping with the assumption made in many services that they will be accessed by relatively powerful clients using high bandwidth, low latency network connections. This assumption, which manifests itself as rich content or low-latency transactions associated with the service, is at odds with the low-bandwidth networks and resource-constrained portable devices used by mobile clients. In our example above, the service may, by default, provide a stream with video size and resolution higher than can be delivered to a user's laptop over a wireless network. Further complicating the situation is the fact that as above, a mobile user might encounter very different connection characteristics over time, ranging from relatively high-bandwidth wireless LAN in a constrained office-like environment, to a metro-area network in populated urban areas, to a cellular network elsewhere. Ideally, the user's interactions with the service would continually adapt to the capabilities of his device and connection.

Unfortunately, such mismatches are poorly handled by current infrastructures, which either provide differentiated service to mobile users or rely upon a close coupling between the service and client applications to adapt to changing network conditions. The first approach, used by several popular news, e-mail, and stock trading services, has the service providing mobile users with different presentation, content, and features than that available to those accessing the service via higher bandwidth connections. Because mobile users are grouped into a small number of classes, they

may not seamlessly receive performance commensurate with the current capabilities of their device or network. This is particularly true in dynamic environments with big variations in available bandwidth (e.g., a wireless LAN user who is at different distances from an access point). The second approach, exemplified by automatic stream selection mechanisms in commercial media players, increases the burden on the application developer limiting its general applicability. Additionally, restricting adaptation to only the end points can often yield sub-optimal behavior. For example, a media player application that switches to a lower quality stream because of congestion in the middle of the network might have been able to continue playing the higher quality stream if the stream was rerouted along a different path.

This paper describes a different approach to resolving the mismatch problem, enabling seamless mobile access to services in resource limited and dynamically changing network environments. Our approach, embodied in the **C**omposable **A**daptive **N**etwork **S**ervices (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 mobile user's connection with the network service making it *network-aware*. CANS supports flexible mapping of these components to path resources, for instance allowing their creation on the user's end device, a proxy server located close to the wireless access point, or an edge server acting as the gateway into the general Internet. 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 two or three network hops, CANS can also be used in wide-area overlay networks to achieve increased control over the entire network path.

Other researchers have also recently proposed similar programmable network infrastructures [1, 2, 5, 7, 25, 28], however CANS distinguishes itself by striving to create such network-aware paths completely *automatically* and additionally support their *dynamic reconfiguration*. To achieve this goal, CANS relies on three key mechanisms:

- A high-level integrated *type-based specification of components and network resources*, which enables late binding of components to paths, essential for flexibility. Unique to CANS is its expression of network characteristics in the same type framework, e.g., network links are represented as components that transform the type of data passing across them. This allows us to reduce the problem of finding appropriate components for given network conditions into one of finding a type compatible sequence.
- An *automatic path creation strategy* based on a polynomial-time dynamic programming algorithm, which simultaneously finds a type-compatible sequence of components that transform the data type produced at the service into a type that can be consumed by the client device, and maps these to underlying network resources so as to optimize a global metric (e.g., client throughput or response time).
- System support for *low-overhead dynamic path reconfiguration*, which includes restrictions on component interfaces and efficient protocols that leverage these restrictions to support three different reconfiguration semantics: no continuity, continuity at the level of *semantic segments*, and full continuity.

We have developed a prototype Java-based implementation of the CANS infrastructure. This prototype is used in the paper to evaluate the effectiveness of our approach, both in terms of the capabilities and performance of the constructed paths as well as the overheads associated with path creation and reconfiguration. We report on experiments conducted using two representative applications, web access and image streaming with client preferences of reduced response time and increased throughput respectively, for multiple network and end-device characteristics reflecting typical mobile use situations. Our results validate the CANS approach, verifying that (1) automatic path creation and reconfiguration is achievable and does in fact yield substantial performance benefits; and (2) that despite their flexibility, both path creation and reconfiguration can be supported with low run-time overhead.

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

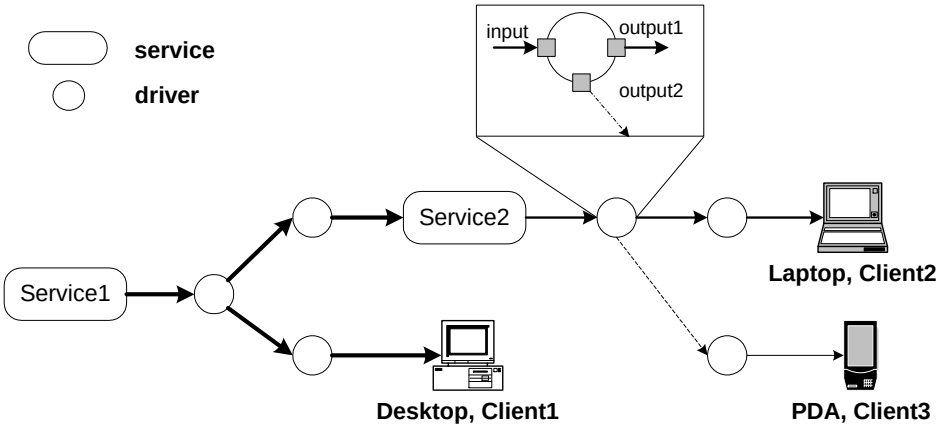


Figure 1: Logical view of a CANS network showing data paths constructed from typed components.

2 CANS Architecture

2.1 Logical View of a CANS Network

CANS views network environments as consisting of client *applications* and network *services*, connected by *data paths*. CANS extends the notion of a data path, traditionally limited to data transmission between end points, to include application-specific components which are dynamically injected by end services, applications, or other entities. Besides supplying consumable data to client applications, these components cooperate to continually adapt the data path to properties of the underlying network and end devices (see Figure 1).

CANS data paths are created dynamically, based on information about user preferences, properties of services and client applications, as well as characteristics of the underlying platform. The components which constitute a data path, the interconnections amongst them, and their internal configuration parameters can all be modified at run time. Modifications are triggered based on either system events (e.g., detection of bandwidth drop on a network link) or component-initiated events (e.g., delayed arrival of data frames). The CANS infrastructure provides support to efficiently reconfigure data paths, while preserving application specific semantics of the transmitted data.

2.1.1 CANS Components

CANS components are self-contained pieces of code that can perform a particular activity on input data, for example, protocol conversion or data transcoding. Network-aware access paths are formed by connecting components with each other based upon compatibility of output and input types (see Section 3 for details). Components come in two flavors: mobile soft-state objects called *drivers* and stateful *services*.

Drivers serve as the basic building block for constructing adaptation-capable, customized data paths. Drivers are standalone *mobile* code modules that perform some operation on the data stream. However, to permit their efficient composition and dynamic low-overhead reconfiguration of data paths, drivers are required to adhere to a restricted interface. Specifically,

1. Drivers consume and produce data using a standard *data port* interface, called a `DPort`. `DPort`s are associated with type information (see Section 3 for details) and distinguished based on whether they are being used for input or output.
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 port is checked for data or an input port receives data.
3. Drivers consume and produce data at the granularity of an integral number of application-specific units, called *semantic segments*. These segments are naturally defined based on the application, e.g., an HTML page or an MPEG frame.
4. Drivers contain only *soft state*, which can be reconstructed simply by restarting the driver. Examples of such state include cache data, compression dictionaries, etc.

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. This achieves nearly the same efficiency, modulo indirect function call overheads, as if driver operations were statically combined into a single procedure call. The last two properties, semantic segments and soft state, enable low-overhead data path reconfiguration and we defer their complete description to Section 5.2.

Services are the second core CANS component. Unlike drivers, which must follow a constrained interface, services can export data using any standard internet protocol (e.g., TCP or HTTP), encapsulate more heavyweight functions, process concurrent requests, and maintain persistent state. The different interface requirements of drivers and services stem from the observation that most current services distributed in the internet are legacy in nature: their source code is general unavailable, and rewriting or modifying them is impractical. The price paid for not adhering to a standard interface is that unlike driver migration, CANS does not explicitly support service migration; a service individually determines how it manages its own state transfer. This design choice reflects the view that services are migrated infrequently and doing so requires protocols that are difficult to abstract cleanly.

CANS provides applications with a general platform to create, compose, and control services across the network. A service is required to register itself by providing a *delegate object* that can control the service and act on its behalf in interactions with the rest of CANS.

2.2 Physical View of a CANS Network

The CANS network is realized by partitioning the service and driver components belonging to multiple data paths onto physical hosts, connected using existing communication mechanisms. The current version of the CANS infrastructure does not address trust and security issues; we assume that hosts underlying the CANS network either belong to the same network administration domain, or are under the control of the same organization. Examples in the first category include campus-area hybrid wired and wireless networks in large shopping centers, office buildings, university, hospital, and airport settings, while those in the second include geographically-distributed content distribution networks with a final wireless hop. Although most mobile use scenarios are likely to benefit from components being deployed in the last two or three hops of the network, CANS can also be used in wider area networks to achieve increased control over the entire network path.

2.2.1 Execution Environment

The Execution Environment (EE) serves as the basic node run-time environment and is responsible for:

- downloading component code from a specified location and instantiating it as required.
- maintaining information about deployed drivers and managing data path operations both within and across EEs, including inserting new drivers, creating new services, and reconfiguring existing paths based upon input from the planning module described in Section 4.
- supporting system- and component-level event propagation within and across EEs (details below).
- interacting with external resource monitors to obtain information about system conditions such as CPU load, network connections, and bandwidth available to a data path. We currently use simple heuristics to estimate network information, however the CANS architecture can easily incorporate resource monitoring tools developed by other researchers [3, 16, 17].

Communication Adapters Inter-EE data transmission is accomplished by auxiliary CANS components called communication adapters, which transmit data *physically* across the network to connect drivers that span different nodes. To achieve this, these components expose a driver-like `DPort` interface. Communication adapters also support two additional kinds of logical connections: (1) between client applications and drivers; and (2) between a driver and a service that exports data using an interface other than `DPort`.

To provide the above functionality, adapters establish point-to-point physical communication links between application wrappers (see below), execution environments, and services. Multiple logical connections can be multiplexed on a single physical link; the latter can exploit transport mechanisms best matched to the characteristics of the underlying network. Communication adapters can additionally encapsulate behaviors that permit them to adapt to and recover from

minor variations in network characteristics. For instance, in nodes supporting both wired and wireless network connections, these adapters can be written to automatically reconnect with an upstream adapter using whichever connection is currently available.

Event Propagation Exchange of control information between components within the same EE or across EEs is accomplished using events.

Components can raise events (tagged with the event name and source) to local registered listeners. This mechanism is used, for example, by the EE to notify interested components about changes in system conditions. The EE also supports a distributed event mechanism, permitting events to be raised on remote EEs. This basic mechanism enables construction of flexible event delivery mechanisms. An example of this is support for *path level events* in CANS: components wanting to communicate control information with other components along the path raise an event that is caught by a local *path event delegate*, which in turn fires a distributed event to its counterpart on the next EE along the data path. This delegate listens for path events arriving through the network, and in response raises a local event that is caught by the local components belong to that path.

2.2.2 Support for Legacy Applications

The CANS infrastructure supports both CANS-aware and CANS-oblivious client applications. The former just hook into the driver and service interfaces described earlier. The latter require more support but are easily integrated because of our focus on stream-based transformations on the data path. Our solution relies on an *interception layer* that is transparently inserted into the application and virtualizes its existing network bindings. The interception layer is injected using a technique known as API interception [13], which relies on a run-time rewrite of portions of the memory image of the application.

The interception layer provides the application with an illusion of a TCP socket, which can be bound to various interfaces (CANS or native network) for actual data transmission. This binding in turn is influenced by an application specific policy, which responds to events (such as connect requests) delivered to it by the interception layer. Thus, enabling CANS support for a new legacy application would require only writing a specific policy for that application. Finally, although our current implementation virtualizes the TCP layer, the technique can as easily support other well-known protocols, such as HTTP.

2.3 Example: A Streaming Media Application

To demonstrate how the CANS infrastructure can enhance mobile user experience in dynamic resource limited environments, we describe a simple example modeled after the teleconference scenario described earlier. Consider a mobile user with a laptop capable of both wired and wireless operation who connects to an Internet-based server to access a media stream. This user starts off at his office desk but then has to leave in the middle to go elsewhere in the building. Let us assume that the user wishes to continue viewing the stream using the laptop's wireless connection, while retaining the same privacy guarantees (freedom from eavesdroppers) he might have had on a wired connection even if, as we assume here, the wireless link has inadequate security.

To ensure seamless mobile access to the network service, an ideal infrastructure would provide the user with good stream quality when he is using a wired connection and that degrades gracefully depending on his distance from the wireless access point. Additionally, the infrastructure would isolate the user from the switch between wired and wireless connectivity and transparently provide the required privacy guarantees. Unfortunately these requirements cannot be satisfied by any current infrastructure: media player applications capable of adapting stream quality to network bandwidth cannot mask the reconnection event, and mobility-aware transport protocols [22, 24] are incapable of adjusting stream quality in any intelligent fashion. Neither set of solutions can satisfy the user's privacy requirements.

The CANS infrastructure successfully enables this scenario by augmenting the path between the user and media server with the following six components: *reconnector(src)*, *reconnector(dest)*, *padder*, *splitter*, *encryption*, and *decryption*. The *reconnector(src)* and *reconnector(dest)* components cooperate to buffer and retransmit frames of the stream, ensuring that the client application always receives a semantically valid frame, even when there is data loss in the switch between the wired and wireless connections. Note that, since, in general, it is impossible to mask the time delay involved in the reconnection, the infrastructure also needs to isolate this delay from the media player application. The *padder* component helps with this, "filling in" legal media frames whenever its input stream stops. The *splitter* component can split the incoming media stream into its video and audio portions, enabling adaptation in low-bandwidth situations.

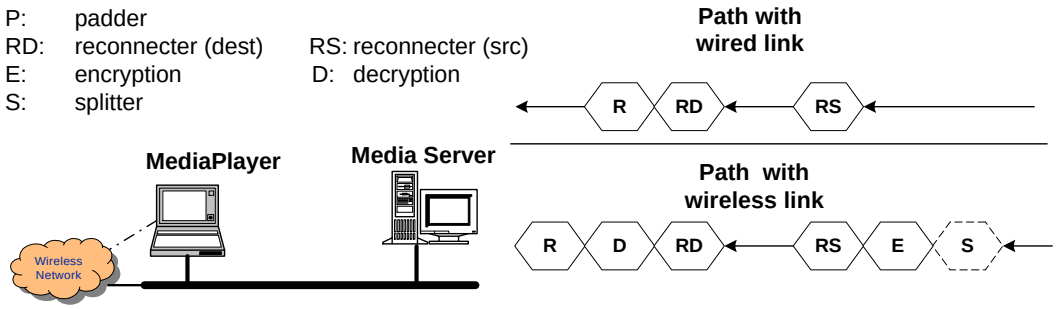


Figure 2: CANS enables seamless mobile access to a network media server by constructing data paths customized to different network conditions.

Finally, the *encryption* and *decryption* components cooperate to maintain privacy of stream data by encrypting it before the wireless link and decrypting it before delivering it to the application.

Thus, these components permit construction of a data path customized to the network conditions encountered during each stage of the example:

- (with the wired link) The data path consists of the *reconnector(src)—reconnector(dest)—padder* sequence, and is capable of isolating the client application from detecting a future disconnection.
- (with the wireless link) The *encrypter—reconnector(src)—reconnector(dst)—decrypter—padder* sequence, when link capacities are sufficient to transmit video+audio to the client. When not enough capacity is present, the sequence would also need to include the *splitter* component at a location determined by the bottleneck link.

While it should be clear by this stage that CANS-like infrastructures can provide substantial flexibility for adapting to resource limited and dynamic network conditions, several concerns need to be addressed before such infrastructures can be widely used. Can such infrastructures *automatically* construct appropriate data paths, while respecting constraints imposed by the network (e.g., that the encrypter and decrypter components need to be at either end of the wireless link)? Can component sequences be mapped to network resources to optimize performance metrics, while respecting node and link capacity constraints? Can data paths be *dynamically* reconfigured whenever system conditions change? Can data paths be constructed and reconfigured without negatively impacting the performance of the system?

The CANS infrastructure includes three key mechanisms—type-based specification of components and network resources, automatic path creation strategy, and system support for dynamic path reconfiguration—which answer each of these questions in the affirmative. We describe these mechanisms in the next three sections and evaluate their effectiveness in Section 6.

3 Type-Based Specification of Components and Network Resources

To automatically construct data paths from a set of components, the first question that must be answered is which application-specific components can be composed together. We formulate this composition problem as a *type compatibility* problem. Central to this formulation is the notion that all data flowing along a data path is *typed*, and that this type is affected both by components along the data path as well as network resources making up the route. In the rest of this section, we describe in turn the type-based representation of components and network resources.

3.1 Representing Component Properties

The composability of CANS components (both drivers and services) is decided by compatibility of type information associated with the input and output ports being connected. The types used in CANS integrate two closely related concepts: *data types* and *stream types*. An additional notion of *data type ranks* helps capture application-specific composition constraints.

Data types are the basic unit of type information, represented by a type object that in addition to a unique type 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, for instance MIME types to represent common media objects.

Traditional mechanisms such as 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 CANS type for MPEG data, which contains attributes for defining the frame size. An `MPEG` type can be defined compatible with another `MPEG` type as long as the former's frame size is smaller than the latter's, 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 CANS drivers operating upon a typed data stream. Stream types are constructed at run time, and are represented as a *stack* of data types. Operations allowed on stream types include *push*, *pop*, *peek*, and *clone*, which have the standard meanings.

Each CANS component with m input ports and n output ports defines a function, which maps its input stream types into output stream types: $f(\mathrm{T}_{\mathrm{in}_1}, \mathrm{T}_{\mathrm{in}_2}, ..., \mathrm{T}_{\mathrm{in}_m}) \rightarrow (\mathrm{T}_{\mathrm{out}_1}, \mathrm{T}_{\mathrm{out}_2}, ..., \mathrm{T}_{\mathrm{out}_n})$ where $\mathrm{T}_{\mathrm{in}_i}$ is the required stream type set for the ith input port, and $\mathrm{T}_{\mathrm{out}_j}$ is the resulting stream type produced on the jth output port. The type compatibility between an input and an output port, which determines whether two components can be connected, 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.

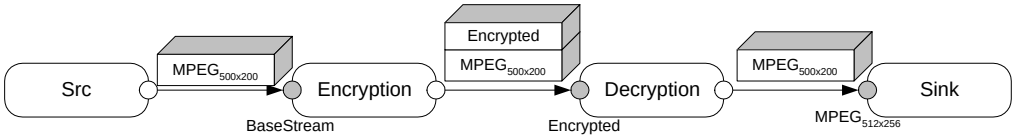


Figure 3: A simple example of component type compatibility.

Figure 3 shows an example of the type compatibility scheme. The source produces MPEG data at resolution 500×200, which needs to be supplied to the sink that can consume MPEG data at resolution 512×256 after going through two components that respectively encrypt and decrypt the data. The figure shows the data types on each of the ports as well as the stream types on the connections. To consider an example, the *Encryption* driver accepts data type `BaseStream` and pushes an `Encrypted` type object onto the incoming stream type. The output port of *Src* is compatible with the input port of *Encryption* because the `MPEG` type object extends the `BaseStream` type. Similarly, the output port of *Decryption*, whose effect is to pop the `Encrypted` type from its incoming stream type, is compatible with the input port of *Sink* because of a type-specific compatibility operator for the `MPEG` type that looks at the resolution attributes.

Figure 3 also highlights the composition advantages of representing stream types as a stack of data types. If components were just modeled as consuming data of a particular type and producing data of another, it would be difficult to express the behavior of the *Encryption* and *Decryption* drivers in a way that permits their use with generic stream types *without* losing information about the original stream type at the output of the *Decryption* driver. In this case, determining whether the *Decryption* driver's output port is compatible with the input port on *Sink* would require examining the entire data path. In contrast, our stream type representation permits local decision making, enabling run-time adaptation via dynamic component composition.

Data type ranks help express application-specific constraints on how CANS components can be composed together by requiring that only types of monotonically increasing ranks can be stacked into a stream type. For instance, by giving the encryption type a higher rank, we can ensure, for any CANS data path requiring both encryption and compression, that encryption always happens after compression. Similarly, the ranking scheme can express that lossy compression can happen after lossless compression but not vice versa, and as in the web access application described in Section 6, that image resizing to reduce bandwidth requirements of web page delivery be employed only after image quality filtering.

To simplify use of CANS in mobile user scenarios, our infrastructure predefines certain common data types. These types are partitioned into different classes such as encryption types, compression types, image transcoding types, etc. Each class is assigned a range of rank values, capturing common constraints of the kind described above. New types required for the application can be easily integrated into this rank hierarchy: the application developer first picks a class in which to put the new data type and then chooses a rank from the class range. The chosen rank must satisfy certain rules, for example, if type t_s is a subtype of t then rank of t_s should not be lower than that of t. We have found this linear organization of the type space to be sufficient for most applications; however, we are currently extending it using

a rule-based mechanism on top of type classes that allows the system to automatically place the types in a rank lattice.

3.2 Representing Network Resource Properties

Network resource characteristics can introduce additional constraints affecting both which components must be present along a data path and how these can be composed. To revisit the example described in Section 2.3, the risk of packet interception on the wireless link necessitates the presence of the encryption and decryption drivers to preserve privacy. Similarly, the padder component is required because it is not possible to bound jitter when the user switches between the wired and wireless networks. Since these drivers are not required if one just examines the type properties of the data 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 after type-compatibility based selection. While this approach works, it compromises on optimality because of poor or redundant placement of these required components.

In contrast, our approach unifies both type compatibility and network resource characteristics in the same framework. We restrict our attention to network links in the following discussion, but the same principle extends to other network resources. The basic idea of our approach is to represent link requirements implicitly by modeling how links affect the types of data that go across them.

To capture the effect of link properties on data types, we introduce the notion of an **augmented type**: each data type is extended with a set of link properties that can take values from a fixed set such as security (used here to denote transmission privacy), reliability, and timeliness, etc. Network links are modeled in terms of the same properties 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 HTML data over an insecure link. Our type framework captures this as follows: the data type produced at the source is represented by `HTML`(*secure*=`true`), the network link is represented by the property *secure*=`false`, and the effect of the link property *secure* on the `HTML` data type by the rule that the augmented type `HTML`(*secure*=`true`) is modified to `HTML`(*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 to isolate others below them in the stream's type stack from having their properties be affected by a link. 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.

3.3 Type-based Modeling of the Streaming Media Application

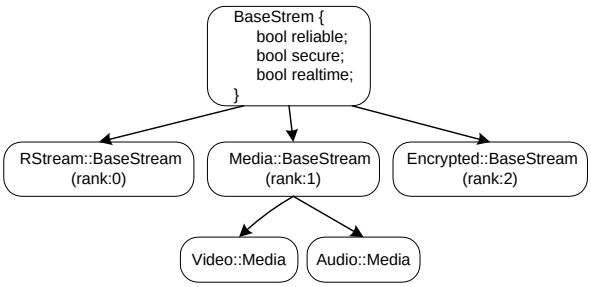
To permit automatic data path construction for the example application described in Section 2.3, the specification of components need to include the following four pieces of information: data type definitions, network links modeled in terms of a set of link properties, rules governing how data types are modified by links, and component properties described in terms of input and output types.

Figure 4(a) shows the data type definitions. `BaseStream` is the basic stream type with three boolean link properties: *reliable*, *secure* and *realtime*. `RStream`, `Media`, and `Encrypted` extend the `BaseStream` type, representing reliable, media, and encrypted streams respectively. `Video` and `Audio` are two subtypes of the `Media` type. The `RStream` type is given a lower rank as compared to the other types to capture an application-specific composition constraint involving the *encryption*/*decryption* and *reconnecter* drivers.

Figure 4(b) shows properties of the wired and wireless links. The wired link is modeled with *reliable* and *realtime* properties set to `false` to capture the fact that it can get disconnected during the access. Similarly, the wireless link has the *secure* property set to `false` to denote its limited support for transmission privacy.

Figure 4(c) shows how these link properties affect different types. "Effect isolation" refers to a type isolating the effect of a link property for data type instances below it in the stack of types making up a stream type. For example, the security property of the `Encrypted` type is unaffected when data of such type traverses an insecure link. Moreover, the type isolates this effect for all of the wrapped types.

Figure 4(d) lists the input/output types of the six components described in Section 2.3, along with the types produced



(a)

	properties		
	secure	reliable	realtime
wired	T	F	F
wireless	F	T	F

(b)

	secure		reliable		realtime	
	T	F	T	F	T	F
Media	—	F	—	F	—	F
RStream	—	F	T*	T*	—	F
Encrypted	T*	T*	—	F	—	F

—: no change *: Effect isolation

(c)

components	Input & output type
Media player (sink)	media:{ realtime = T; reliable = T; secure = T; }
source	video:{ realtime = T; reliable = T; secure = T; }
splitter	audio, video
padder	media:{ realtime = T; } → media:{ realtime = *; }
encryption	Encrypted * ← *
decryption	* ← Encrypted *
reconnecter (src)	RStream * ← *
reconnecter (dest)	* ← RStream *

(d)

Figure 4: Types in the streaming media example: (a) data type definitions; (b) link properties; (c) effect of link properties on augmented types; and (d) input and output types of components.

by the source and that required by the sink. To consider some examples, the sink specification says that the client application requires a reliable, real time, and secure `Media` type. The *padder*, which fills in legal frames whenever it does not receive input in a timely fashion, is represented as a component that transforms the input type `Media` with an arbitrary value for the *realtime* property, into the output type `Media` with *realtime*=`true`. Similarly, the *encryption* component is modeled as an entity that converts an arbitrary stream type at its input into a new stream type consisting of the `Encrypted` type wrapping whatever was originally present. The *decryption* component performs the reverse operation, stripping away the `Encrypted` type out of the stream type.

The primary advantage of modeling component properties in a type framework is that all legal data paths associated with a given set of network conditions correspond simply to type-compatible component sequences that transform the source data type into that required by the sink. The important point here is that these legal sequences can be inferred fully automatically. In this example, the two network conditions of interest are whether the user connects to the server using a wired link or a wireless link.

With the wired link, the above type specifications yield the following two legal sequences: *reconnecter(src)—reconnecter(dest)—padder*, and *splitter—reconnecter(src)—reconnecter(dest)—padder*. Informally, the former might be used when link capacities are sufficient for transmission of the original video+audio stream to the client, while the latter is required when this is not the case.

With the wireless link, again we have two sequences: *encryption—reconnecter(src)—reconnecter(dest)—decryption—padder*, and *splitter—encryption—reconnecter(src)—reconnecter(dest)—decryption—padder*. Notice that the *encryption* and *decryption* components are required to preserve the secure property of a stream transmitted across the wireless link (see Figure 4(c)). Note also that an alternate type-compatible component sequence *reconnecter(src)—encryption—decryption—reconnecter(dest)—padder* is disallowed because of the ranks associated with the `RStream` and `Encrypted` types.

Having obtained legal sequences, the next step towards automating data path construction is choosing one of them and mapping it to underlying network resources to optimize some global performance metric.

4 Automatic Path Creation Strategy

The CANS path creation strategy embodied in the EE *planning module* 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.

In the following part of this section, we first describe the single path creation strategy and then discuss the reservation scheme to handle multiple paths. Details of the path reconfiguration procedure are deferred to Section 5.

4.1 Single Path Creation

Creation of a single data path 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 can be viewed as the shortest path problem in the node graph, which takes into consideration bandwidth on links between nodes in different domains and the relative loads on nodes within the same domain. Given the large amount of literature available on similar problems, we will not discuss this further.

The component selection process takes as input the augmented type at the data source, the augmented type required at the sink, and the selected route (whose links may transform augmented types as described earlier). In this paper, we restrict our attention to single input, single output components; i.e., all selected plans consist of a sequence of components. Most of the application scenarios we have experimented with fall into this category.

The heart of our strategy is a dynamic programming algorithm to simultaneously select components and map them to the route in a fashion that optimizes overall throughput, while ensuring a lower bound on data quality. Other optimization metrics could also be used such as minimal latency, best data quality, etc. We first describe a base version of the algorithm where only simple (non-stacked) data types are present and network resources *do not* affect the type of data crossing them, and then discuss how this base algorithm is extended to handle the more general case of stream types and link properties.

4.1.1 Base Algorithm

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

A **driver component** d is modeled in terms of its *computation load factor*, $\mathrm{load}(d)$, and its *bandwidth impact factor*, $\mathrm{bwf}(d)$. $\mathrm{load}(d)$ captures the average per-input byte cost of running the component, while $\mathrm{bwf}(d)$ reflects the average ratio between input and output bandwidths. For example, a compression component that reduces stream bandwidth by a factor of two has a $\mathrm{bwf} = 0.5$. Similarly, for the corresponding decompressor, $\mathrm{bwf} = 2.0$.

A **data path**, $D = \{d_1, \ldots, d_n\}$, is a sequence of type-compatible components, as defined in Section 3. A **type graph** formalizes this notion: vertices in the graph represent types, and edges represent components that can transform the type of one vertex to the type of the other. There might be multiple edges between two vertices in the type graph; the degree of any vertex obviously does not exceed the total number of components in the system.

A **route**, $R = \{n_1, n_2, \ldots, n_p\}$, is a sequence of nodes obtained using the route selection algorithm. $R(n_i, n_j)$ refers to the subsequence starting at node n_i and ending at node n_j. Each node n_i is modeled in terms of its *computation capacity*, $\mathrm{comp}(n_i)$, which represents the number of operations that the node can perform every unit time. A link between two nodes, l_{ij}, is modeled in terms of its bandwidth, $\mathrm{bw}(l_{ij})$. Both $\mathrm{comp}(n_i)$ and $\mathrm{bw}(l_{ij})$ are defined in terms of route resources available for a particular path.

A **mapping**, $M : D \rightarrow R$, associates components on data path D with nodes in route R. We are only interested in mappings that satisfy the following restriction: $M(d_i) = n_u, M(d_{i+1}) = n_q \Rightarrow u \leq q$; i.e., components are mapped to nodes in path sequence order. This is a reasonable assumption for data paths crossing multiple networking segments which have different properties.

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 data path D that transforms t_s to t_d and can be mapped to R to yield maximum throughput.

The problem as stated above is NP-hard. To make the problem tractable, we view the computation capacity as partitionable into a fixed number of *discrete* load intervals; i.e., capacity is allocated to components only at interval granularity. Not only is this assumption practical, but it also 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]). For this algorithm, we are interested only in a subset of all possible vectors that have the pattern $\{1, \ldots, 1, r_i, 0, \ldots, 0\}$ for reasons explained below. It can be easily verified that the total number of such legal vectors is $p \times L$, where p is the number of nodes and L is the number of the discrete load intervals.

Dynamic Programming Strategy

The intuition behind the algorithm is to construct, for different amount of route resources, optimal mappings for data paths with increasing numbers of components, say $k+1$, using as input optimal partial solutions involving k or fewer components. The form of the legal resource vector above is explained as follows: when we decide to assign the $k+1$th component d_{k+1} in the data path to a node n_i in the route, the drivers before d_{k+1} cannot make use of the nodes after n_i because of our mapping definition. Thus, we need only consider partial solutions for resource vectors that take the form $\{1, \ldots, 1, r_i, 0, \ldots, 0\}$.

More formally, the algorithm builds up partial optimal solutions, $s[t, \vec{A}, k], \forall t, \vec{A}, k$, where each such solution yields maximum throughput for transforming the source type t_s to an arbitrary intermediate type t, using a data path with k components or fewer and requiring no more resources than $\vec{A}$. The dynamic programming strategy defines how these solutions can be constructed in a bottom up fashion:

- Step 1 solutions simply consist of zero-component paths (edges in the type graph) that transform t_s into an arbitrary intermediate type t, and require no more than $\vec{A}$ resources (for each $\vec{(A)}$ for a given route R).
- Assume that Step $k-1$ solutions have been constructed. These consist of optimal paths of $k-1$ or fewer components that transform the source type into all intermediate type while using no more than $\vec{A}$ resources (for any $\vec{A}$). The dynamic programming step works as follows.
- To construct a step k solution for a given type t and resource vector $\vec{A}$, consider all possible intermediate types t' that can be transformed to t; i.e., all those types for which an edge $d = (t', t)$ is present in the type graph. For each such t', consider all possible mappings of the associated component d 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 $\mathrm{load}(d)$), combine this component with the optimal Step $k-1$ solution $s[t', \vec{A'}, k-1]$. Note for each mapping where $M(d) = n_i$, we set $r'_k = 0 \; \forall k > i$ in $\vec{A'}$. The combined mapping that yields the maximum throughput is deemed the optimal Step k solution.

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

Two additional points need some clarification. First, in the above algorithm, we need to know how much resources to set aside for component d before we can combine d with an optimal Step $k-1$ solution. The problem here is that d's resource requirements $\mathrm{load}(d)$ are expressed in terms of per-input byte costs, and are difficult to evaluate without knowing what the input bandwidth is, which itself is only known once the Step $k-1$ solution is selected. We break this cyclic dependency by first *guessing* the resource requirement of d and then evaluating the throughput for this guess. The guess that yields the maximum throughput is picked to reflect d's resource usage. Note that because of discretized load levels, we only need to make a constant number of guesses at each step and moreover, one of these must yield the optimal solution.

Second, in order to ensure the strategy optimizes throughput *while* ensuring a lower bound on quality, before committing to a Step k solution, the algorithm must verify that the lower bound on quality is satisfiable in the unresolved portion of the data path. Note that to resolve this question, we simply need partial solutions precomputed by running the algorithm in the "reverse" direction, i.e., starting from t_d, with a different objective, that of maximizing data quality.

The algorithm terminates at Step $k_{\max} = p \times n$, where p is the number of nodes and n is the number of components. This follows from the observation that for real components, there is no throughput benefit from mapping multiple copies of the same component to the same node. The solution $[t_d, \vec{A}_{\max}, k_{\max}]$, if present, yields the optimal selection and mapping of components to transform t_s to t_d along route R. The complexity of this algorithm is $0(n^3 \times p^3)$ as opposed to $O(p^n)$ for an exhaustive enumeration strategy. As stated earlier, in most mobile access scenarios benefiting from CANS, p is expected to be a small constant, with overall complexity determined by the number of components.

4.1.2 Extension 1: Handling Stream Types

Stacked stream types complicate the type graph used in the base algorithm because of a need to represent each of the stream types that can be generated by a stackable driver component. For instance, a *Zip* driver can consume data of both `HTML` and `WML` types producing different stream types, which must be separately represented. The naive approach of explicitly enumerating each stream type in the graph (e.g., `Zip-HTML` and `Zip-WML` in the example) does not scale because the size of the type graph grows exponentially with the number of simple data types.

We employ two strategies to ensure that the type graph does not become intractably large. First, we restrict the type graph to include only those stream types that are reachable from the source data type, and which in turn can reach the destination type required by the client. This reduces the number of enumerations by a large amount because of the observation that the total number of possible stacking operations involving a specific type is limited. Second, we exploit the data type *ranks* described in Section 3.1, which impose constraints on component composition and thereby reduce the number of stream types that will be constructed. In the web access application described in Section 6, the combination of these two strategies reduces the number of type graph nodes to just 3 when creating a path for the data type `mime/text` and 6 when creating one for the type `mime/image`. In contrast, the application drivers repository included about 20 simple types that would have resulted in a substantially larger number of nodes otherwise.

4.1.3 Extension 2: Dealing with Network Resource Properties

The algorithm as described so far does not consider the possibility of network resources affecting stream data types. Taking network links as an example, to cope with their effect on stream types, the algorithm needs to incorporate two modifications. First, the type graph is now defined in terms of augmented types, making explicit the differences between streams that have the same data type but different values of link properties. Because both the number of such properties as well as the set of values associated with each property are expected to be small, such enumeration results in only a small increase in the size of the type graph.

The other modification is to the recursive step in the dynamic programming algorithm described above. In particular, when developing Step k solutions, the optimal Step $k-1$ solution that is combined with the selected one-component partial mapping must take into account possible type translations because of an intermediate link. In other words, for a given intermediate type t', we now need to consider solutions $s[t'', \vec{A}', k-1]$ where t'' is translated by one component and one link (if it exists in the corresponding mapping) into t'. This modification does not change the overall complexity of the algorithm.

4.2 Management of Multiple Paths

To concurrently support multiple data paths over shared resources, we employ a strategy called *adjustable reservation*. The basic idea is very simple, each network resource (nodes, links, etc.) is partitioned into a number of equal *shares* (see Figure 5). Individual data paths are provisioned by reserving node and link shares along the selected route, using the single path creation strategy described earlier to deploy path components. Share enforcement is achieved using a user-level sandboxing strategy [4], which provides control over CPU and network resource utilization. Note that after allocation of shares, each of the paths can be created and maintained independently. For scalability, the controllers for individual paths are distributed across the entire system.

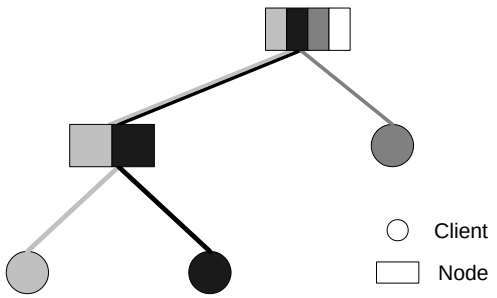


Figure 5: Multiple paths are hosted on a shared set of resources by allocating shares of resources to each individual path.

Node and link shares are autonomously adjusted whenever the aggregate load on a resource changes, either because data paths are deleted or new data paths need to be created, or because of external conditions such as link congestion. Individual path controllers are informed of these adjustments using events, possibly triggering the reconfiguration procedure described in the next section. To balance between the need to efficiently use a resource and avoid potentially high-overhead share adjustments, the partitioning procedure is discrete. To take an example, consider a resource that is initially partitioned into four shares, all of which are being used by active data paths. When a fifth path needs to be created, the resource is repartitioned to have eight shares (as opposed to five). This strategy trades off overall resource utilization versus repartitioning frequency.

Our current scheme uniformly divides up shared resources amongst active paths. This can be easily extended, as in network provisioning literature [10,11], to accomodate other schemes such as a weighted partitioning or even guaranteed provisioning of route resources.

5 System Support for Efficient Path Reconfiguration

Data paths may 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. In this section, we first describe the reconfiguration semantics supported by CANS, and then the required system support.

5.1 Reconfiguration Semantics

The central question about reconfiguration is what can the application assume about data in transit or buffered within components when 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. Applications involving non-critical data (e.g., news feeds) can exploit in-order delivery guarantees to perform efficient recovery.
- **Level 2** semantics provides the guarantee of delivering complete *semantic segments*, essentially simplifying the task of the application recovery code. Semantic segments represent application-specific notions of a useful granularity of data. For example, in a streaming media application, a semantic segment might correspond to individual frames. Level 2 semantics ensure that a frame is either completely delivered or not delivered at all.
- **Level 3** semantics provide full continuity guarantees with exactly-once semantics, completely isolating the application from the fact that the path has been reconfigured. 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. An example is the *padder* component of the media streaming application described in Section 2.3.

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 data path reconfiguration and error recovery strategies that rely upon retransmission at the granularity of semantic segments.

Note that this property only refers to the logical view of the driver, and admits physical realizations that transmit data at any convenient granularity as long as segment boundaries are somehow demarcated (e.g., with marker messages).

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. For example, a Zip driver that produces compressed data will produce semantically equivalent output (i.e., uncompressed to the same string) if presented with the same input strings.

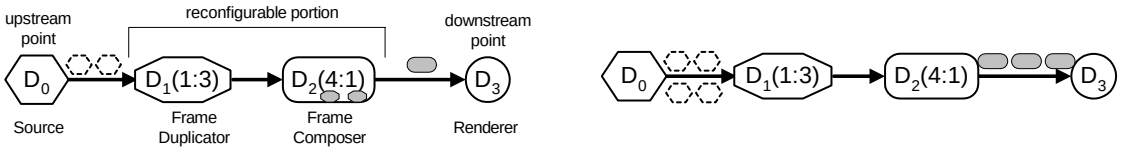


Figure 6: An example of data path reconfiguration using semantics segments.

Together, these two properties enable low-overhead path reconfiguration as described below.

5.3 Reconfiguration Protocol

Path reconfiguration is triggered by events generated either by the EE, which relays a change in network resource characteristics or available path resources to the path controller, or by a component that detects a change in an application-specific quality metric. The reconfiguration process consists of three major steps: (1) generation of a new selection and mapping of components ("plan" for short) by the path controller; (2) ensuring required semantics prior to freezing data transmission; and (3) deploying the new plan and resuming data transmission. Step 1 uses the planning algorithm described earlier, optionally reusing some of the partial solutions constructed during initial deployment, and can be overlapped with ongoing transmission. Step 3 involves a standard two-phase commit like procedure to synchronize reconfiguration activities among multiple nodes along the path. We describe Step 2 in additional details below.

Step 2 requires slightly different support for the three reconfiguration semantics described earlier. Since activities for Levels 1 and 2 are a subset of that for Level 3, our description focuses on the latter. The underlying problem is that to maintain semantic continuity and exactly-once semantics, any scheme must take into account the fact that the portion of the data path being reconfigured can have stream data that has been partially processed: in the internal state of drivers, in transit between execution environments, or data that has been lost due to failures. Note that the soft-state requirement on its own does not provide any guarantees on semantic loss or in-order reception.

Figure 6 shows an example highlighting this problem. To introduce some terminology, we refer to the portion of the data path that needs to be reconfigured because of network changes (failures are an extreme example) as the *reconfiguring portion*, and the components immediately upstream and downstream of this portion with respect to the data path as the *upstream point* and *downstream point* respectively. In the example, driver d_0 is a source of MPEG data, driver d_1 is an MPEG frame duplicator which produces 3 frames for each incoming frame, driver d_2 is an MPEG frame composer which generates one MPEG frame upon receiving four incoming frames from d_1, and d_3 is a renderer of MPEG data. The reconfiguring portion consists of drivers d_1 and d_2. Consider a situation where system conditions change after the upstream point d_0 has output two frames, and the downstream point d_3 has received one frame. At this point, the portion containing d_1 and d_2 cannot be reconfigured because doing so affects semantic continuity. It is incorrect to retransmit either the second segment from d_0 whose effects have been partially observed at d_3, or the third segment, which would result in a loss of continuity at d_3.

The reconfiguration protocol leverages the semantic segments and soft state restrictions placed on driver functionality as follows. Intuitively, the first restriction allows us to infer which segments arriving at the downstream point of the reconfiguring portion depend on a specific segment injected at the upstream point and vice-versa, while the second makes it always possible, even if any internal driver state is reset, to recreate the same output segment sequence at the downstream point by just retransmitting selected input segments at the upstream point. Our solution exploits these characteristics to provide the required guarantees by just combining *buffering* and *delayed forwarding* of semantic segments at the upstream and downstream points respectively, with *selective retransmission* of segments that are incompletely delivered. The correspondence between upstream and downstream segments is completely determined by driver characteristics in the reconfigurable portion; the implementation just needs to track marker messages that demarcate segment boundaries.

This scheme uniformly handles both the situation where drivers continue error-free operation but the data path needs to be reconfigured in response to system conditions, as well as the situation where link or node errors cause partial driver state to be lost; the difference in the two situations is only whether the protocol is executed on demand or always. For the first situation, we defer reconfiguration to the time when the system can guarantee continuity and exactly once semantics

for Level 3 (respectively, complete delivery of a semantic segment for Level 2). Upon receiving an event that triggers reconfiguration, the upstream point starts buffering segments while continuing to transmit them, in effect flushing out the contents of intermediate drivers. The downstream point monitors the output segments arriving there, waiting until it completely receives an output segment from upstream satisfying the property that *all subsequent segments correspond only to input segments at the upstream point that are either buffered or not yet transmitted*. For Level 2 semantics, one need only wait for the simpler requirement that all semantic segments that originate from the same input segment are delivered. At this time, the system can be stopped and the reconfigurable portion replaced by a semantically equivalent set of drivers. To restart, the upstream point retransmits starting from the first segment whose corresponding output segment was not delivered.

In our example, reconfiguration works as follows (assuming Level 3 semantics). To start with, the upstream point (d_0) starts buffering every segment it sends out after this time. When the downstream point (d_3) receives a complete upstream segment (in this case this happens when the third segment output by d_2 is received), it raises an event. The path controller can now freeze d_0, and replace d_1 and d_2 with a compatible driver graph. To restart, d_0 retransmits starting from segment 5. In this case d_3 does not need to discard anything. Error recovery on this portion requires d_0 to buffer its output segments and have the downstream point pass on segments to d_3 only in units of 3 segments at a time.

6 Performance Evaluation

To evaluate the effectiveness of CANS mechanisms detailed in Sections 3–5 in enabling automatic creation and reconfiguration, we measured the performance and overheads of automatically created network-aware data paths in mobile usage scenarios in the context of two applications, web access and image streaming.

We describe in turn our experimental platform and an analysis of the effectiveness of automatic path creation and reconfiguration. All experiments reported here were conducted using a Java-based prototype of the CANS infrastructure. An early version of this prototype is available for download from `http://www.cs.nyu.edu/pdsg/projects/cans`

6.1 Experimental Platform

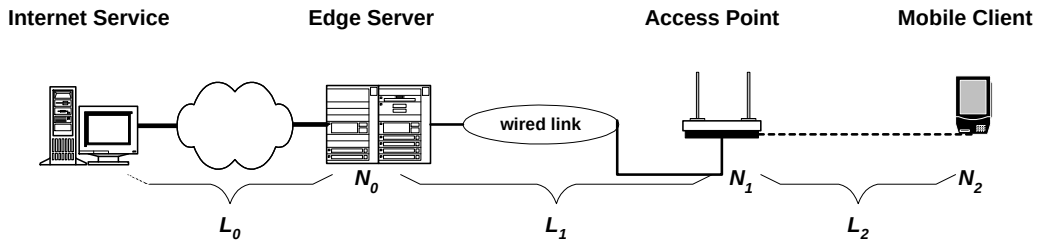


Figure 7: A typical network path between a mobile client and an internet services.

For all of our experiments, we consider a typical network path between a mobile client and an Internet server as shown in Figure 7. This platform models a mobile user using a portable device (N_2) such as a laptop, pocket PC or cellular phone to access network services. 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).[1]

The **web access application** consists of a browser client and transcoding components that reduce download times under low-bandwidth network conditions by dynamically compressing text and/or degrading image quality. Previous research has shown that such an approach is effective [5, 20]. In this paper, we focus on the question of whether an

[1]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.

appropriately customized subset of these components can be automatically deployed to minimize download time for different network conditions.

The **image streaming application** consists of a simple downloadable applet that sets up a connection with a server, receiving and displaying images periodically pushed by the latter. This application is representative of news feeds and tickers on many financial web pages. For our application, we require that images are available at the client within a certain time deadline, and that the transmission is private.

Component	*Input/Output Types*	*Load* (ops/byte)	*Bandwidth Factor*
ImageFilter	**F**: Image $\longrightarrow$ Image	1.64×10^{-6}	3.92
ImageResizer	**R**: Image$\longrightarrow$Image	8.335×10^{-6}	3.92
Zip	**Z**: * $\longrightarrow$ ZipType/*	1.3×10^{-7}	3.15
Unzip	**U**: ZipType/*$\longrightarrow$*	1.2×10^{-7}	0.32
Demultiplexer	**D**: MIME $\longrightarrow$ Image,Text	negligible	1.0
Multiplexer	**M**: Image,Text $\longrightarrow$ MIME	negligible	1.0
Encrypter	**E**: * $\longrightarrow$ Encrypted/*	4.35×10^{-6}	1.0
Decrypter	**D**: Encrypted/* $\longrightarrow$ *	4.35×10^{-6}	1.0

Table 1: Characteristics of components employed in the web access and image streaming applications.

Table 1 lists the characteristics of components used in the two applications. The *ImageFilter* and *ImageResizer* components degrade image quality and the *Zip* and *Unzip* components work together to compress text pages as required. *Demultiplexer* and *Multiplexer* enable different CANS paths for text and images, and the *Encrypter* and *Decrypter* components help guarantee transmission privacy. The load values shown in Table 1 are normalized with respect to a Pentium III 1 GHz machine, which is assumed to have a computing power of 1 ops/second. 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 24 KB JPEG images for the first application, and a 24 KB JPEG image for the second. All experiments used the same data inputs that the components were profiled on. This is a simplifying assumption, but reasonable given our primary focus 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 a topic deferred to future research.

6.2 Effectiveness of Automatic Path Creation

To model different network conditions likely to be encountered along a mobile access path, we defined twelve different configurations listed in Table 2. 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.[2] These configurations are grouped into three categories, based on whether the mobile link L_2 exhibits cellular, infrared, or wireless LAN-like characteristics. Four 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 [4]. As before, the computation power of different nodes is normalized to a 1 GHz Pentium III node.

Table 2 also identifies, for each platform configuration, the plan automatically generated by CANS for the web access application. The plans themselves are shown in Figure 8. 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. Contrast this plan with plan A (for configurations 1 and 2) where the proxy server now contains both an *ImageFilter* and an *ImageResizer* component. The latter is required because the bandwidth reduction due to just the *ImageFilter* component is insufficient to cross the 19.2 Kbps link.

Figure 9 shows the performance advantages of the automatically generated plans when compared to the response times incurred for direct interaction between the mobile client and the server (denoted Direct in the figure). The bars in Figure 9 are normalized with respect to the best response time achieved on each platform (so lower is better). In all

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

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**

*Experiment conducted on real (as opposed to "sandboxed") hardware.

Table 2: Twelve configurations representing different loads and mobile network connectivity scenarios, identifying the CANS plan automatically generated in each case.

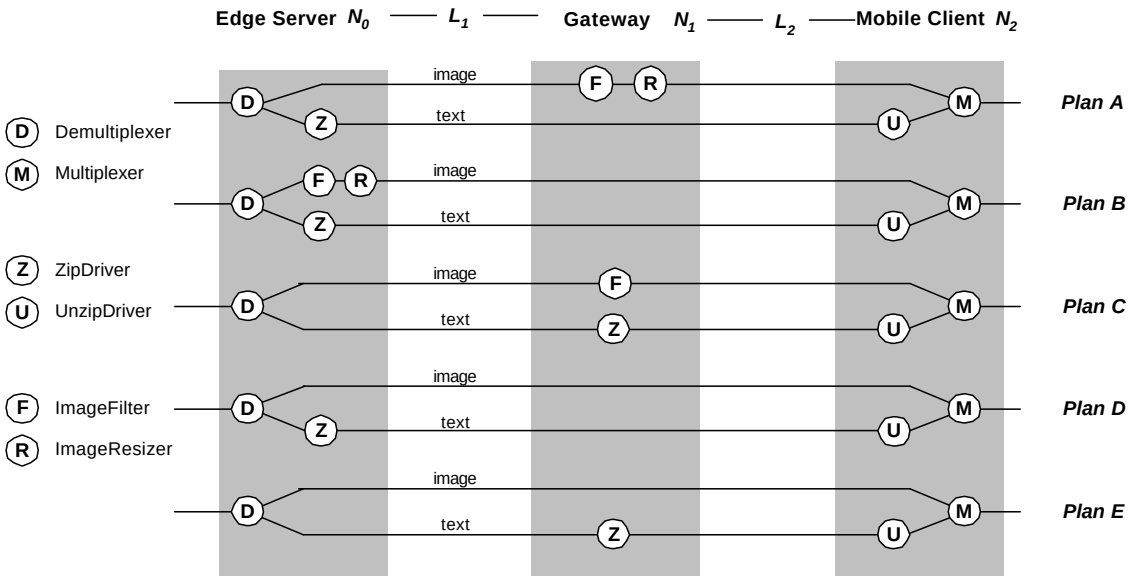


Figure 8: Component placement for the five automatically generated plans.

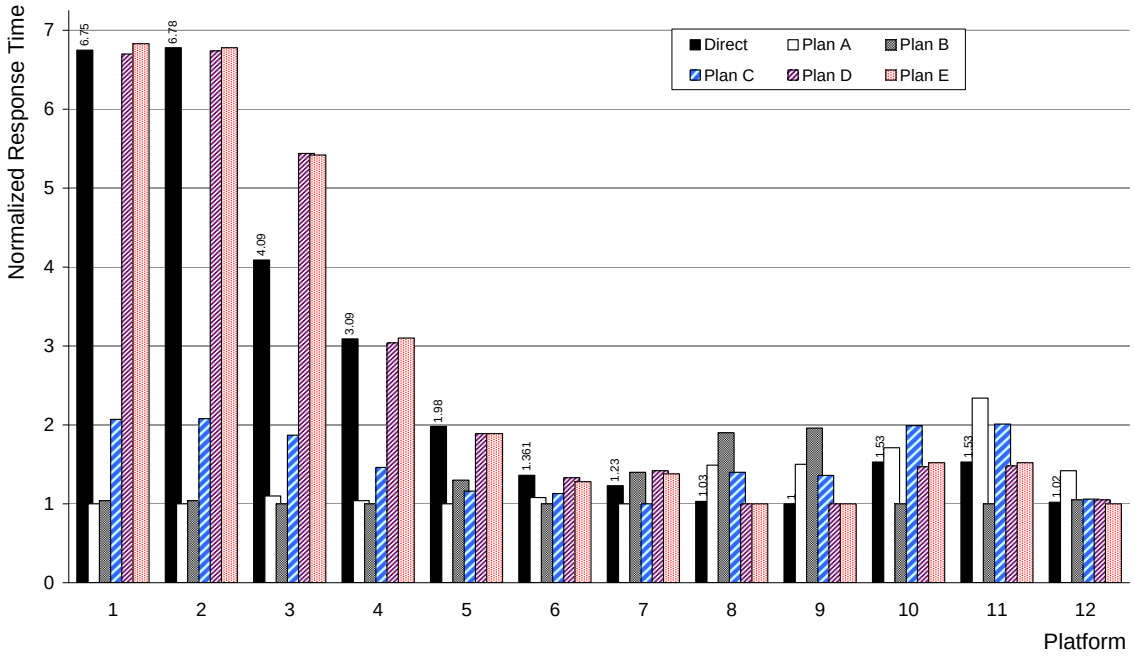


Figure 9: Response times achieved by different plans for each of the twelve platform configurations compared to that achieved by direct interaction. All times are normalized to the best performing plan for each configuration.

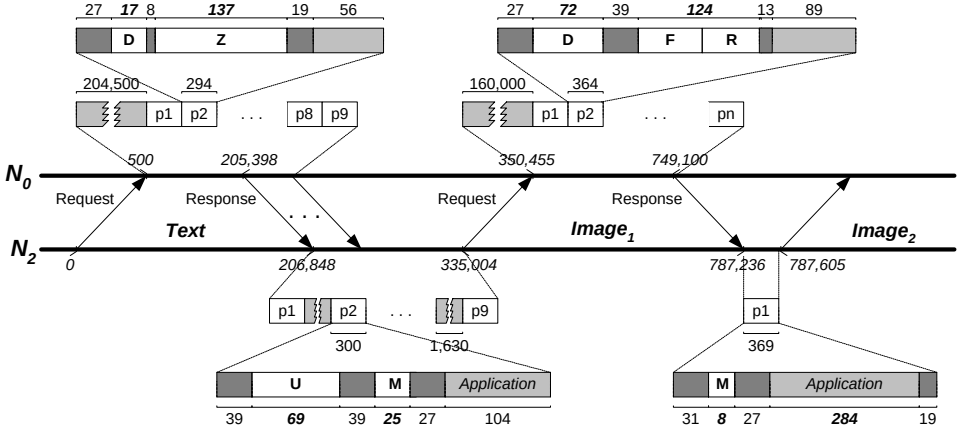


Figure 10: Timeline of requests and responses for plan B running on platform configuration 10 (all times are microseconds). The blocks marked **D**, **M**, **Z**, **U**, and **F** correspond to the executions of the respective components. Communication overheads, including wait times, are shown using gray, whereas CANS overheads are shown using hatched blocks. *Application* refers to the overhead of communicating the data to the client application.

twelve configurations, the generated plans improve the response time metric, by up to a factor of seven. In addition, these improvements would further increase if there were a web caching node along the route. Note that 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 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.

It is interesting to note that CANS achieves substantial performance improvements despite run-time overheads on the critical path. To understand whether other applications with different component characteristics would yield similar improvements, we profiled our implementation to construct a timeline of the operations involved in processing a client request for the web page. Figure 10 shows the overall timeline for plan B running on platform configuration 10, and breaks down portions of this timeline into individual operations performed by the CANS execution environment and the components themselves for processing a single text and image packet. The original client request results in the downloading of the text portion of the page, and is followed by requests for each of the six contained images. A text request is received by the edge server N_1, which forwards it to the web server and waits for the latter to respond. Text responses comprise several packets, each of which passes through the *Demultiplexer* and Zip drivers on the edge server, and the Unzip and *Multiplexer* drivers on the client before being delivered to the browser application. Similarly a response to an image request comprise multiple packets, each of which flow through the *Demultiplexer*, *ImageFilter*, and *ImageResizer* drivers on the edge server and the *Multiplexer* on the client before being delivered to the application.

The timeline shows that for this application, CANS overheads are negligible and dominated by the round-trip between the edge server and the web server (0.2 seconds on the text path and 0.16 seconds on the image path). Even if this were not the case, CANS overheads (shown hatched in the figure) for retrieving data from the network and supplying it to each driver in turn are small for all but very fine-grained components (the *Demultiplexer* and *Multiplexer*). For the components used in this study, CANS incurs an average cost of about 25μs per driver invocation, and we expect these overheads to improve significantly as the system is tuned for performance.

6.3 Performance of Data Path Reconfiguration

To evaluate the effectiveness of our path reconfiguration approach, we ran the image streaming application under dynamically changing network conditions, letting the CANS infrastructure automatically generate and reconfigure its access path. For this application, the three levels of reconfiguration semantics correspond to no guarantees about continuity (Level 1), the guarantee that the application only sees complete images (Level 2), and that the application sees no semantic information loss (Level 3). The base network configuration corresponds to Platform 7 in Table 2, with two changes introduced 25 seconds and 50 seconds into the experiment. The first change degraded the bandwidth between the client and the access point ($L2$) to 440 Kbps from the original 1 Mbps. The second change modeled the transition of the network from (secure) wired connectivity to (insecure) wireless connectivity as in the example scenario described in Section 2.3. Since our focus was on measuring the overheads of the reconfiguration procedure, our experiment had an external procedure generate the necessary events and coordinate with the "sandbox" code to control bandwidth available to the application.

Figure 11 shows the paths created by the planning procedure in response to the events (top left), and how the reconfiguration procedure transitions among these paths along the execution timeline (top right). The initial path **A** contains only an *ImageFilter* component running on the proxy server. The first event, triggered when bandwidth drops, results in the introduction of an additional component, the *ImageResizer*, on the proxy server (path **B**). Note that CANS reconfiguration is accomplished completely automatically, without any involvement from the application code. Depending on the semantics that need to be supported, the total time for reconfiguration is either 0.65 seconds (for Levels 1 and 2) or 1.01 seconds (for Level 3). The path again gets reconfigured when the second event is received, corresponding to the switch between wired and wireless connectivity; the new path **C** now contains *Encrypter* and *Decrypter* components on the proxy server and client nodes to ensure secure transmission. As before, reconfiguration is achieved automatically, but incurs slightly larger overheads (0.74 seconds for Levels 1 and 2, and 1.15 seconds for Level 3), because of the additional work involved in orchestrating reconfiguration activities across multiple nodes.

To better understand the contributing factors, we broke down the 1.01 seconds required for Level 3 reconfiguration into four stages (bottom part of Figure 11): (1) construction of a new plan and computing the delta from the current plan; (2) send command to the upstream and downstream points to start buffering and monitoring; (3) waiting for the

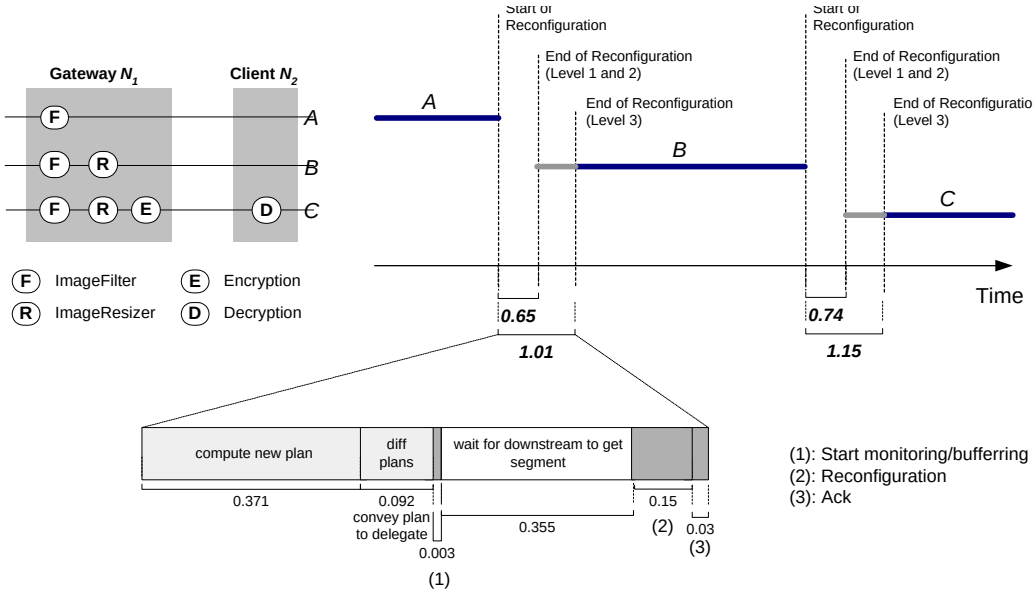


Figure 11: Path reconfiguration in the image streaming application. All times are in seconds.

reconfiguration condition(eg. next complete semantic segment arrival) to become true; and (4) the two-phase procedure to install the new path and resume data transmission. Note that stage 3 is the time spent waiting for the server to send out the next semantic segment, thus should not be counted as the overhead of reconfiguration. This stage can be bypassed for a given reconfiguration semantics level, if it is possible to infer that the reconfiguration condition is immediately satisfied (for this application, stage 3 can be bypassed for both Level 1 and Level 2). The cost breakdown shows that the dominant contributors are plan creation (0.46 seconds) and waiting for the next semantic segment to arrive (0.35 seconds), with reconfiguration protocol steps (gray blocks) incurring very small overhead (0.18 seconds). It should also be noted that although the total reconfiguration time is around 1 second, data keeps flowing downstream during the first 3 stages of reconfiguration. For the first reconfiguration instance, data transmission need only be frozen for 0.18 seconds (4th stage). Even this time can be hidden from the client application employing client-side components such as the *padder* in Section 2.3.

7 Related Work and Discussion

7.1 Related Infrastructure

The research described in this paper is very closely related to several recently proposed infrastructures that aim to augment the traditional notion of a network path with injected application-specific components. The application-specific functionality can be introduced only at the end-points or could be distributed on intermediate nodes. Odyssey [21], Rover [14] and InfoPyramid [18] are examples of systems that support end point adaptation. Each system provides only minimal support for composing adaptation activities across multiple nodes, and consequently may not be flexible enough to cope with changes in intermediate links.

The cluster-based proxies in BARWAN/Daedalus [5], TACC [6], MultiSpace [8], ICAP [9] and OPES [12] are examples of systems where application-transparent adaptation happens in intermediate nodes (typically a small number) in the network. Active Services [1] extends these systems to a distributed setting by permitting a client application to explicitly start one or more services on its behalf that can transform the data it receives from an end service. A different perspective is offered by systems such as Conductor [28], which automatically deploy multiple application-transparent adaptors along the data path between applications and end services. Although such systems retain backward compatibility with existing applications, the lack of application input limits their flexibility. Furthermore, such systems rely upon self-describing properties of data streams, a condition that may or may not hold given increasingly proprietary content.

7.2 Path Creation Strategy

The Ninja project's Automatic Path Creation (APC) service [7] can be used to create paths between various end devices and services. Both APC and our approach formulate the component selection problem in terms of type compatibility, however, there are significant differences. At a high level, unlike the performance-oriented focus of our work, APC is a function-oriented method, which ignores network link properties and node and link resource constraints. A consequence of this difference is that a shortest-path approach to planning suffices for Ninja (with the restriction that a data type can appear only once along a path), while we need a more sophisticated dynamic programming-based approach. Other differences include our support for path reconfiguration and a more general notion of data, stream, and augmented types, which were motivated by a desire to model link characteristics in a unified fashion and contrast with Ninja's notion of a relatively simple string type.

Kiciman and Fox [15] have proposed a general path infrastructure framework for composing mediators distributed across a network of machines. This infrastructure builds upon Ninja's APC service and suffers from the same limitations. Furthermore, this approach separates out logical path creation (choice of components) from the mapping of components to physical resources. As we have shown in Section 6, such decoupling can produce suboptimal solutions because of poor or redundant component placement.

Recent work in the Scout project [19] has looked at a template based path construction algorithm for delivering media objects that takes into consideration the latter's resource requirements, user preferences, node capabilities, and programmer-provided path rules. This work shares its performance focus with ours, however, the primary difference arises from the fact that unlike our high-level type-driven approach, here a programmer must a priori construct path templates and store them into a central database. The Scout algorithm takes a lower-level approach, simply choosing an appropriate template and instantiating it based on other programmer-provided rules that decide whether or not a component can be created on a resource. We avoid this last problem because of the application-level nature of our components, which rely on a relatively standard execution environment interface (the Java virtual machine in our case). On the flip side, the Scout approach does a better job of modeling low-level resource properties such as the availability of a specific kind of video hardware or NIC.

The Panda project [23] also proposes a planning scheme for optimally placing network-level components to modify an application's data stream in response to unfavorable network conditions. While two schemes are discussed, one based upon selection from a reusable plan set and the other based on exhaustive constraint space-based search, to the best of our knowledge these schemes have not yet been implemented or evaluated with real applications.

7.3 Standardization Efforts

Our work is also complementary to emerging standards for efficient content delivery. The CC/PP (Composite Capabilities/Preference Profiles) protocol from the World Wide Web Consortium (W3C) [27], and the UserAgent protocol from the Wireless Application Protocol (WAP) forum [26] focus on small devices with different user preferences and aims to automate the process of setting up the delivery. The Open Pluggable Edge Services architecture (OPES) [12] and Internet Content Adaptation Protocol(ICAP) [9] aims to define the protocols for a broad set of services which can cooperate with each other to achieve the efficient delivery of complex content over the Internet.

To the best of our knowledge, the approach described in this paper is one of the first schemes to not only consider the functionality of the data path, but also takes both network link properties and node resource constraints into account. Our work is also one of the first to perform a detailed evaluation of the overhead of path creation, and reconfiguration, and measure the performance of the deployed paths. While we expect the performance to improve as the CANS implementation is further tuned, the numbers in this paper provide a concrete baseline for the potential of automatic approaches for constructing network-aware access paths.

We should also note that the limitations of our approach described here. First there is one limitation for CANS data paths. In CANS what a component in data path can do basically is to transform the format of the data while keeping the original semantic information (we are in the same belief as other researchers that it is hard to capture semantics with type mechanism). Also we have not addressed the security issues in mobile code execution in this paper. Lastly the efficiency of our planning algorithm is dependent on the resource monitoring entity which provides the dynamical information for network resources, which currently is an external part of CANS framework. We are currently exploring these issues and will cover them in our future work.

8 Conclusions and Future Work

This paper has presented 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 internet services. This approach leverages a type-compatibility formulation of the problem, which takes as input only high level specifications of component behavior and network route characteristics. Novel to this formulation is the fact that constraints due to node and network link characteristics are naturally integrated into the type model simply by modeling the latter as entities that transform the type of data passing across them. This formulation lends itself to a dynamic programming based polynomial-time algorithm, which simultaneously selects and maps appropriate components to optimize a global metric such as client throughput or response time. This algorithm is complemented by an efficient semantics-preserving data path reconfiguration strategy. Experiments with the planning algorithm and reconfiguration in the contexts of a web access scenario and an image streaming application using the CANS infrastructure under various network and end device characteristics have verified that automatic path creation and reconfiguration is both feasible and can yield substantial performance benefits. Thus, in contrast to current-day static access paths to internet services, our work argues for a flexible approach where paths leading to these services are automatically and dynamically composed to satisfy user preferences and network resource constraints.

CANS is one component of a larger project, Computing Communities, which focuses on distribution middleware for legacy applications. Our future work involves generalizing the CANS planning algorithms to handle multiple simultaneous data paths, the strategy to support efficient reconfiguration for multi-ported components, efficient resource monitoring across networks and integrating CANS with related efforts emphasizing resource management and security issues.

References

[1] E. Amir, S. McCanne, and R. Katz. An Active Service Framework and its Application to Real-time Multimedia Transcoding. In *Proc. of the SIGCOMM'98*, August 1998.

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

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

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

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

[6] A. Fox, S. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based Scalable Network Services. In *Proc. of the 16th ACM Symp. on Operating Systems Principles*, October 1997.

[7] 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.

[8] S. D. Gribble, M. Welsh, E.A.Brewer, and D. Culler. The MultiSpace: An Evolutionary Platform for Infrastructual Services. In *Proc. of the 1999 Usenix Annual Technical Conf.*, June 1999.

[9] ICAP Protocol Group. ICAP: the internet content adaptation protocol. In *http://www.i-cap.org/icap/media/draft-elson-opes-icap-01.txt*, February 2001.

[10] IETF Differentiated Services Working Group. Differentiated services (diffserv). In *http://www.ietf.org/html.charters/diffserv-charter.html*.

[11] IETF MPLS Working Group. Multiprotocol label switching (mpls). In *http://www.ietf.org/html.charters/mpls-charter.html*.

[12] IETF OPES Working Group. Open pluggable edge services. In *http://www.ietf-opes.org/*, 2000.

[13] G. Hunt. Detours: Binary interception of win32 functions. In *Proc. of the 3rd USENIX Windows NT Symp.*, Settle, WA, July 1999.

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

[15] E. Kiciman and A. Fox. Using Dynamic Mediation to Intergrate COTS Entities in a Ubiquitous Computing Environment. In *Proc. of the 2nd Handheld and Ubiquitous Computing Conference (HUC'00)*, March 2000.

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

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

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

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

[20] B. Noble. System Support for Mobile, Adaptive Applications. *IEEE Personal Communications*, pages 44–49, February 2000.

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

[22] C. Perkins. IP Encapsulation within IP. In *RFC 2003*, 1996.

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

[24] Alex C. Snoeren and Hari Balakrishnan. An end-to-end approach to host mobility. In *Proc. 6th International Conference on Mobile Computing and Networking (MobiCom)*, 2000.

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

[26] WAP. WAP Forum Specifications. Technical report, http://www.wapforum.org/what/technical.htm, January 2000.

[27] W3C CC/PP Workgroup. CC/PP specification. Technical report, http://www.w3c.org, August 1999.

[28] 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.