CANS: Composable, Adaptive Network Services Infrastructure

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

Abstract

Ubiquitous access to sophisticated internet services from diverse end devices across heterogeneous networks requires the injection of additional functionality into the network to handle protocol conversion, data transcoding, and in general bridge disparate network portions. Several researchers have proposed infrastructures for injecting such functionality; however, many challenges remain before these can be widely deployed.

CANS is an application-level infrastructure for injecting application-specific components into the network that focuses on three such challenges: (a) efficient and dynamic composition of individual components; (b) distributed adaptation of injected components in response to system conditions; and (c) support for legacy applications and services. The CANS network view comprises *applications*, stateful *services*, and *data paths* built from mobile soft-state objects called *drivers*. Both services and data paths can be dynamically created and reconfigured: a planning and event propagation model assists in distributed adaptation, and a flexible type-based composition model dictates how new services and drivers are integrated with existing components. Legacy components plug into CANS using an interception layer that virtualizes network bindings and a delegation model.

This paper describes the CANS architecture, and a case study involving a shrink-wrapped client application in a dynamically changing network environment where CANS improves overall user experience.

1 Introduction

The emergence of new networking technologies such as broadband to the home, Wireless 3G [18], and Bluetooth [10], coupled with increasing numbers of network-capable end devices holds the potential for future application services that significantly enhance user experience by providing seamless, ubiquitous access. To take an example, consider the following scenario. Alice, a telecommuting employee, starts her day by initiating a teleconference on her laptop connected to the internet using a wired LAN. During the conference, a hub failure renders the wired LAN unavailable. Fortunately, the service detects this, and seamlessly switches data transmission to a local wireless network while simultaneously degrading picture quality upon recognizing that the wireless LAN has insufficient bandwidth for continuous video at the original resolution and rate. Shortly after, Alice leaves her office to meet a client. She shuts down her laptop, and resumes the teleconference in her car using a PDA connected to a metro-area wireless network. The service further downgrades the media stream (say to only include audio), while recording the full stream at a server that Alice can check offline.

Although the above scenario is compelling, its requirements — rapid creation and deployment of new services, application-aware computation in the network, and dynamic and distributed adaptation — are poorly handled by current internet infrastructure. Moreover, the existing view which hides network characteristics from the application and treats services as standalone entities is incompatible with the large variation in network and end-device characteristics. Current-day data paths can include links with very different bandwidth, delay, and error characteristics, ranging from serial links to wireless to broadband to fiber. Hiding these differences from the application will result in unsatisfactory application performance, and the alternative of providing differentiated service for different networks/end-devices cannot adequately cope with dynamically changing environments.

One solution to these problems is to inject additional functionality into the network that can dynamically adapt to resource characteristics of end-devices and network links by handling activities such as protocol conversion, data transcoding, etc. Several researchers have proposed infrastructures for achieving this goal, ranging from end-point solutions [12, 15] to more distributed alternatives that introduce application-aware functionality either at the network level [17, 3, 20] or at the application level [1, 6, 8]. Although these systems have articulated

a common set of high-level architectural requirements, many challenges, particularly with respect to dynamic services management and composition, remain before the infrastructures see widespread deployment.

This paper describes **C**omposable **A**daptive **N**etwork **S**ervices (CANS), an application-level infrastructure for customizing the data path between client applications and services, which focuses on three such challenges:

- *Efficient and Dynamic Composition*, enabling separately defined components to be dynamically instantiated and interconnected using efficient mechanisms (e.g., shared memory within a host).
- *Dynamic and Distributed Adaptation*, enabling adaptation to environment changes along the entire data path while incurring low overhead and maintaining overall application semantics.
- *Support for Legacy Applications and Services*, enabling the latter to be integrated into CANS with minimal effort. Requiring rewrites of each application and service is neither practical nor desirable.

CANS addresses these challenges by constructing networks that include *applications*, stateful *services*, and *data paths* between them built up from mobile soft-state objects called *drivers*. Drivers implement a standard interface, permitting efficient composition and semantics-preserving adaptation. Both services and data paths can be dynamically created and reconfigured: a planning and event propagation facility enables distributed adaptation, and a flexible type-based composition model dictates how new services and drivers are integrated with existing ones. CANS provides three adaptation modes to permit cost-functionality tradeoffs: intra-component, by reconfiguring data paths, and by creating new services and data paths. Legacy components plug into CANS using delegation and an interception layer that transparently virtualizes network bindings, currently TCP sockets.

CANS has been implemented on Windows 2000 clients and Java/RMI-capable intermediate hosts. Each node runs the CANS execution environment, which supports dynamic creation, migration, and adaptation of drivers and services. Experience with a case study involving a shrink-wrapped application (Windows MediaPlayer) in a dynamically changing network environment indicates the potential of our approach: CANS permits dynamic deployment and distributed adaptation of application-aware components to improve overall user experience.

The rest of this paper is organized as follows. Section 2 presents the CANS architecture, with details about its components and distributed adaptation support appearing in Sections 3 and 4. Section 5 presents the CANS implementation and the MediaPlayer case study. Section 6 discusses related efforts, and Section 7 concludes.

2 CANS Architecture

2.1 The Logical View

CANS views networks as consisting of *applications*, *services*, and *data paths* connecting the two. CANS extends the notion of a data path, traditionally limited to data transmission between end points, to include application-specific components dynamically injected by end services, applications, or other entities; these components adapt the data path to physical link characteristics of the underlying network and properties of end devices (see Figure 1(a)).

Components are self-contained pieces of code that can perform a particular activity, e.g., protocol conversion or data transcoding. Components operate on *typed* data streams and are connected with each other based upon compatibility of output and input types (see Section 3 for details). Injected components come in two flavors: stateful *services* and mobile soft-state objects called *drivers*. Services extend the original data path to multiple hops, and drivers generalize the traditional notion of a data path to include data transformation in addition to transmission. The primary reason for distinguishing between drivers and services is to ensure efficiency.

CANS data paths are created dynamically, using 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., breaking of a network link) or component-initiated events. The CANS infrastructure provides support to efficiently reconfigure data paths, while preserving application semantics.

2.2 The Physical View

The CANS network is realized by partitioning the services and data paths onto physical hosts, connected using existing communication mechanisms. The CANS Execution Environment (EE) serves as the basic runtime environment on these hosts and includes the following functional modules (see Figure 1(b)): *class manager*, *plan manager*, *driver and service manager*, *event manager*, and *resource monitor*.

The class manager handles downloading of component code and instantiation of the components. The plan manager is responsible both for creating the initial plan comprising drivers, services, and data paths in response to a request trapped by the interception layer, as well as replanning in response to system conditions. The driver and service manager maintains information about deployed drivers and manages data path operations, includ-

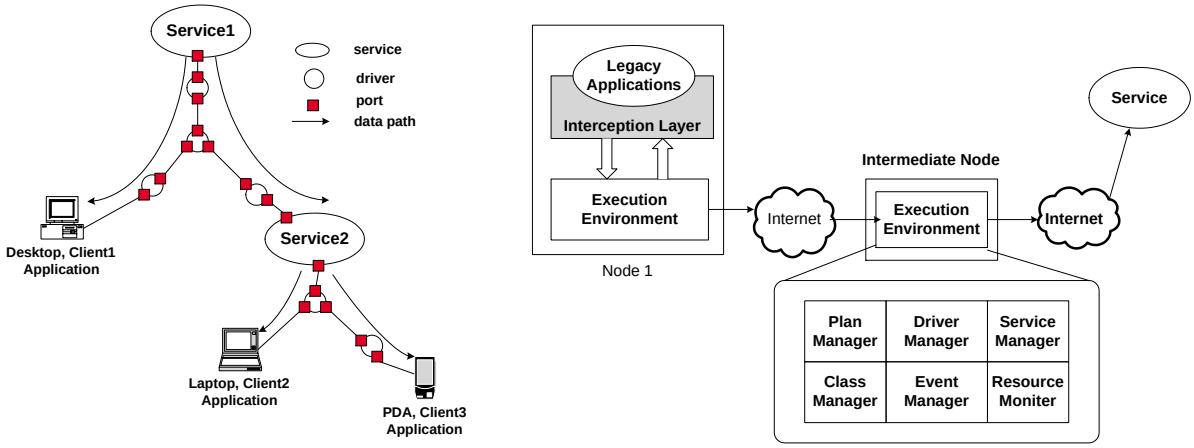


Figure 1. (a) Logical organization of CANS, (b) Physical realization of CANS data paths.

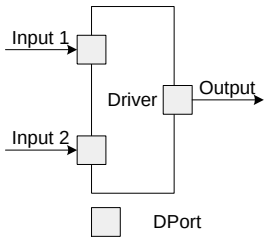
ing inserting new drivers, creating new services, and reconfiguring existing paths as required. The event manager is responsible for receiving both system-level and component-level events and propagating these on to interested components. The resource monitor monitors system conditions such as CPU availability or network interface state, informing the event manager when registered trigger conditions fire.

3 CANS Components

CANS components include drivers, services, and auxillary components that interconnect execution environments, applications, and legacy services.

3.1 Drivers

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 as shown in Figure 2. Specifically,




```
class Driver {
  String driverID;

  TList outTypes(TList inTypes);
  DPortList getPorts();
  DPort getPort(String PortId);
  void push(DInPort input);
  void pull(DOutPort out);
  void raiseEvent(CANSEvent e);
  void registerListener();

}
```

(b)

Figure 2. Driver functionality (a) and interface (b).

1. Drivers consume and produce data using a standard *data port* interface, called a `DPort`. `DPort`s are typed (details below) 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 one of its output ports is checked for data, or one of its input ports 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. 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 (see Section 4.2). 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).

4. Drivers contain only *soft state*, which can be reconstructed simply by restarting the driver. Stated differently, given a semantically equivalent sequence of input segments, a soft-state driver always produces a semantically equivalent sequence of output segments. 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.

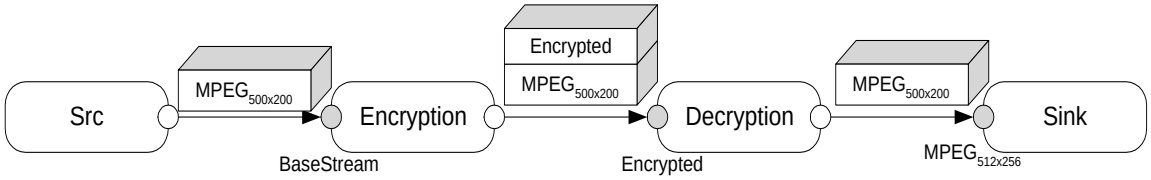


Figure 3. A simple example of type compatibility.

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 semantic segments and soft-state properties enable low-overhead dynamic adaptation, either within a single driver or across data path segments while preserving application semantics. The driver interface (see Figure 2) permits a driver to create and listen to events, facilitating its participation in distributed adaptation activities.

Type-based Composition

The composability of CANS components (both drivers and services) is decided by compatibility of the 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*.

CANS 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. Traditional mechanisms such as type hierarchies can still be used to organize data types; however, our scheme permits flexible type compatibility relationships not easily expressibly 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.

CANS stream types capture the aggregate effect of multiple CANS drivers operating upon a typed data stream. Stream types are constructed at run time, and representable 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:

$$f(T_{in_1}, T_{in_2}, ..., T_{in_m}) \rightarrow (T_{out_1}, T_{out_2}, ..., T_{out_n})$$

where T_{in_i} is the required stream type set for the ith input port, and T_{out_j} is the resulting stream type produced on the jth output port. The type compatibility between an input and an output port, 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 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 affect 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 type, it would be difficult to express the behavior of the *Encryption* and *Decryption* drivers in a way that permits their use for a variety of generic stream types *without* losing information about the original stream type at the output of the *Decryption* driver. Thus, 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, a prerequisite for runtime adaptation via dynamic component composition.

3.2 Services

The second core CANS component are *services*. Unlike drivers that represent rigidly constrained, mobile, soft-state adaptation functionality, 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 identifying the data types it supports, optionally providing a *delegate object* that can control the service and act on its behalf in interactions with the rest of CANS. The delegate object implements a standard interface consisting of activating and suspending the service, and receiving CANS events. Service composition is similar to driver composition, using types supplied at registration time.

3.3 Communication Adapters

Communication adapters are auxillary CANS components, which transmit data *physically* across the network to connect drivers that span different nodes. To achieve this, these components expose the same `DPort` interface, appearing to other drivers just as a regular driver. Communication adapters also support two additional kinds of logical connections: (1) between 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 physical communication links between application wrappers (see below) and execution environments, between two execution environments, and between an execution environment and a service. Multiple logical connections can be multiplexed on this 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, these adapters can be written to use one of several network alternatives, automatically transitioning between them to improve performance. The continuity semantics upon such reconnection are dictated by the requirements of the data types associated with the adapter's ports.

3.4 Support for Legacy Applications

The CANS infrastructure supports both CANS-aware and CANS-oblivious applications. The former just hook into the driver and service interfaces described earlier. The latter require more support but are easily integrable 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 [11], which relies on a run-time rewrite of portions of the memory image of the application.

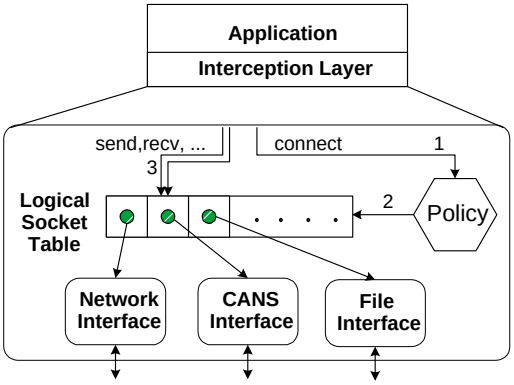


Figure 4. Architecture of the interception layer.

The general architecture of the interception layer is shown in Figure 4. 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. An application specific policy responds to events (such as connect requests) delivered to it by the interception layer, which in turn influences the binding. 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.

4 Distributed Adaptation in CANS

CANS supports three modes of adaptation in response to dynamic changes in system characteristics: (1) *intra-component adaptation*, where each service or driver detects and adapts to minor resource variations on its own; (2) *data path reconfiguration and error recovery*, where

the data path undergoes localized changes involving insertion, deletion, and reordering of drivers; and (3) *replanning*, where existing data paths are torn down and new ones constructed to respond to large-scale system variations. These three modes represent different points on the cost-functionality spectrum, enabling the system to respond to system events with the least overhead possible. To the best of our knowledge, CANS is unique in providing system support for data path reconfiguration.

4.1 Intra-Component Adaptation using Distributed Events

Each CANS driver and service can incorporate its own adaptation behavior that may or may not be coordinated with adaptation in other components. For example, a frame-dropping component can alter its policies upon detecting different levels of back-pressure on its output buffers. Note that adaptation in a single component is completely isolated as long as its effect is restricted to be within a single semantic segment (see Section 3.1).

To trigger adaptation, CANS provides distributed event propagation support, permitting components (including delegate objects for legacy services) to raise arbitrary events as well as listen for specific ones. Event support is realized by a per execution environment **Event Manager**, which is responsible for catching, firing, and transmitting events across the network. Event raising and firing is implemented using simple method calls and callback functions associated with the relevant component.

There are two major types of CANS events: events from the local resource monitor, indicating a change in resource status, and events from components on the data path. The first kind of events are sent only to local components that register themselves as interested listeners. The second kind, issued by components along a data path, are first sent to the *plan event delegate* (see Section 4.3), which is responsible for propagating the event along the data path as well as handling plan-specific events, such as events to trigger replanning.

4.2 Data Path Reconfiguration and Error Recovery using Semantic Segments

Insertion, deletion, or reordering of drivers along an active data path provides great flexibility in responding to a range of resource variations and link/node failure. However, a fundamental problem is that any such reconfiguration must preserve application semantics. In this paper, we focus on maintaining semantic continuity and exactly-once semantics. Specifically, any scheme must take into account the fact that the portion of the data path affected by the reconfiguration can have stream data that has been partially processed: in the internal state of drivers, in transit between execution environments, or

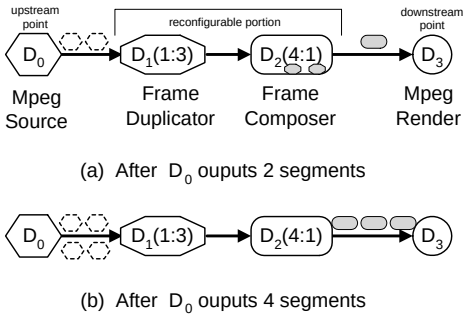


Figure 5. An example of data path reconfiguration using semantics segments.

data that has been lost due to failures. Note that although the soft-state requirement discussed in Section 3.1 permits us to restart a driver, it does not provide any guarantees on semantic loss or in-order reception.

Figure 5 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 a change in system conditions on the physical nodes or links (failures are an extreme example) as the *reconfigurable 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.[1] 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 reconfigurable 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 data path portion containing d_1 and d_2 cannot be reconfigured because doing so affects semantic continuity. The reason is that because of partially processed data in that portion, 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 CANS infrastructure supports semantics preserving data path reconfiguration and error recovery by leveraging two restrictions placed on driver functionality, specifically semantic segments and soft state (see Section 3.1). Informally, the first restriction permits the infrastructure to infer which segments arriving at the

[1] For simplicity, we restrict our description to reconfigurable portions that have exactly one upstream and one downstream point. However, the solution is easily extendable to more general structures.

downstream point of the reconfigurable 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. 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. For the first situation, we defer reconfiguration to the time when the system can guarantee continuity and exactly once semantics. When some CANS events trigger 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 from upstream point either buffered at the upstream point or not yet transmitted*. 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.

The same basic scheme also permits error recovery on portions of the data path that can be tagged a priori as possible sources of failure. The upstream point by default buffers all input segments before passing them on. The downstream point delays passing to the downstream driver any output segments that cannot be reconstructed in their entirity from input segments that are buffered at the upstream point, effectively isolating the downstream drivers from any duplicates that might get produced due to retransmission. When it is safe to pass on an output segment, the corresponding buffered input segments can be discarded. Upon an error, the affected components are re-instantiated, any buffered output segments at the downstream points discarded, and retransmission resumed from the first input segment whose corresponding output segment was never observed by the downstream driver. This scheme can be trivially extended to permit error recovery on portions that include services with checkpoint/restart facilities: the service needs to checkpoint whenever it produces a segment that corresponds to an input segment boundary.

In our example, reconfiguration works as follows:

1. The upstream point (d_0) starts buffering every segment it sends out after this time.
2. When downstream point (d_3) receives a complete segment from the upstream point (in this case this happens the third segment output by d_2 is received), it raises an event to the plan manager.
3. The plan manager can now freeze d_0, and replace d_1 and d_2 with a compatible driver graph.
4. 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.

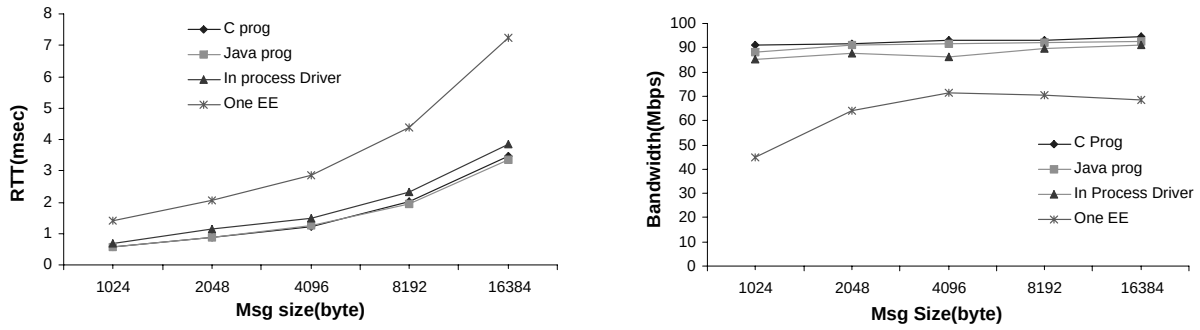
4.3 Planning and Global Reconfiguration

A plan refers to the deployment of drivers, services, and data paths in response to a request from a client application to connect to an end service. The key component responsible for planning in CANS is the *plan manager*, which is triggered when the interception layer detects a connect attempt on a TCP socket of interest. The plan manager takes responsibility both for creating the original plan, as well as changing it as required based on evolving system conditions. Such replanning is a last resort; as stated earlier, most changes are expected to be handled either entirely within a component or through localized data path reconfiguration.

The planning procedure consists of two steps: *route selection* where a graph of nodes and links is selected for deploying the plan, and *driver selection* where appropriate drivers and services are mapped to the selected route. Space considerations prevent us from describing the steps in full detail, so we just highlight the overall strategy restricting our attention to plans that involve a single source and a single sink.

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.

Driver selection bridges source and sink types while (1) efficiently using link and node capabilities along the selected route, and (2) overcoming problems caused by link properties such as insecure transmission and packet loss. The first subproblem amounts to selecting type-compatible components to construct the data path such



(a) Round Trip Time (b) Bandwidth

Figure 6. Latency and bandwidth impact of the CANS infrastructure.

that node and link capacities are not exceeded, and some overall path metric (e.g., throughput) is optimized. The second subproblem imposes restrictions on the stream type at various points in the data path; for example, encrypted data is required in order to cross an insecure link if the sink requires a secure stream.

Our scheme unifies these two subproblems by defining the notion of an *augmented type*: each data type is extended with a set of link properties (e.g., security, reliability, and timeliness) that can take values from a fixed set. 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 of HTML data transmitted over an insecure link, the data type represented by `HTML`(*secure*=`true`) is modified to `HTML`(*secure*=`false`) upon crossing a link with property *secure*=`false`. As a refinement to this base scheme, some data types have the capability to *isolate* others below them in the type stack associated with a stream from having their properties be affected by a link. For example, the `Encrypted` type isolates the *secure* property of types that it "wraps", i.e., encrypted data still remains secure after crossing insecure links.

Thus, the inputs to the driver selection process are 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). We use a dynamic programming algorithm to simultaneously select a component and map it to the route in a fashion that optimizes overall throughput. The partial solutions that make up the algorithm essentially look at the problem of converting the source type to an intermediate type on a subset of the route using only a fixed number of components. The complexity of this algorithm is $O(n^3 \times m^3)$, where n is the number of the components and m is the number of nodes.

5 Experience with Using CANS

We have been experimenting with a prototype CANS implementation on Windows 2000 clients and Java capable intermediate hosts, which currently emphasizes functionality and correctness over performance. Both the execution environment (EE) and driver components are written in Java. The interception layer described in Section 3.4 makes use of the Detours toolkit [11] to divert required application functions by rewriting portions of the memory code image. To set up the plan, the interception layer interacts with the plan manager on a distinguished EE, which in turn builds the plan, partitions it, and downloads plan fragments to individual environments. Interactions between different EEs make use of Java/RMI. Data transmissions between components, which are more performance critical, makes use of the communication adapters described in Section 3.3.

In this section, we first describe microbenchmarks reflecting overheads of using the CANS infrastructure, and then a larger case study that evaluates its flexibility.

5.1 Microbenchmarks

All measurements below were taken on a set of Pentium II 450Mhz, 128 MB nodes, running Windows 2000 and connected using 100 Mbps switched Ethernet.

Figure 6 shows the overheads introduced by CANS, measured in terms of how they impact communication between an application and an end service. Each graph shows the round-trip time and bandwidth achievable for different message sizes for four configurations: C prog and Java prog refer to our baselines, corresponding to application and server programs that communicate directly using native sockets in C or Java respectively. In process Driver and One EE refer to basic CANS configurations; the former shows the case when null drivers and a communication adaptor are embedded into the application interception layer and indicates the basic over-

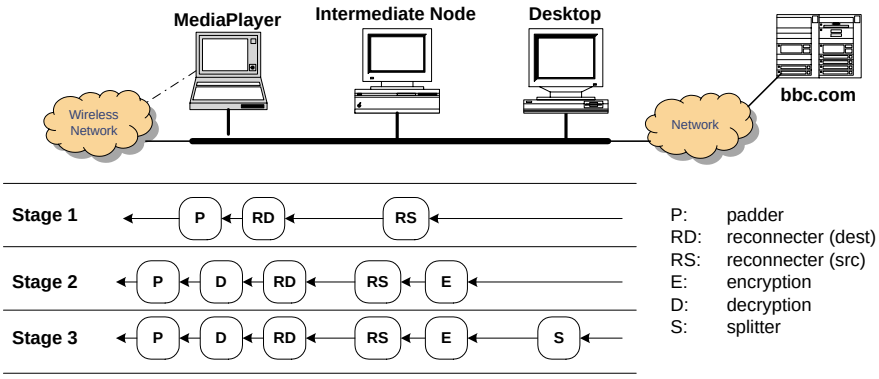


Figure 7. Case study: MediaPlayer with CANS infrastructure and the components added in each stage.

heads of driver composition, and the latter considers the case where the data path includes null drivers on an intermediate host between the application and service.

Figure 6 shows that the In process Driver configuration introduces minimal additional overheads when compared with the Java prog configuration (less than 10% arising from extra synchronization and data copying), attesting to the efficiency of our driver design and composition mechanism. On the other hand, the One EE configuration does show marked degradation in performance, primarily because of context switch costs and the fact that the transmitted data has to traverse across application-level and network-level four times instead of two times. However, given that intermediate EEs are intended to be used across different network domains where other factors dominate latency and bandwidth, this overhead is unlikely to have much overall impact.

5.2 Case Study

To evaluate whether CANS provides enough flexibility to support large-scale applications, we conducted a case study involving a shrink-wrapped application: Microsoft MediaPlayer. Our objective was to see whether CANS could be used to improve user experience with the application in a dynamically changing network environment *without* requiring explicit user participation (see Figure 7). The case study highlights CANS capabilities for (1) automatically selecting and deploying components suited to different network characteristics, driven solely by high-level type specifications at the source and sink, (2) dynamic and distributed event-driven adaptation upon detecting a change in system conditions, and (3) integrating with legacy applications and services.

The experimental environment (see Figure 7) consists of the client application run on a laptop with both wireless and wired network interfaces, a desktop capable of hosting services, an intermediate computer capable of hosting an execution environment, and an internet-based server providing media content (in our case, this was the bbc.com server). The case study consists of three stages: the laptop starts off being connected to the network using its wired interface, then is disconnected from the wired LAN, and finally physically moved away from the wireless access point. The transition from wired to wireless LAN is accompanied by a loss in security properties as well as a drop in bandwidth, which becomes worse in the third stage. CANS seamlessly insulates the client application from all network changes, continuing to seamlessly provide the user with the best experience afforded by underlying network characteristics.

CANS achieves this behavior by dynamically deploying appropriate components from a predefined set according to the planning algorithm described in Section 4.3. Note that route selection in this case is trivial, the one route involving all of the machines shown in Figure 7. The planning algorithm takes as input four pieces of information: the type definitions, the set of components, links modeled in terms of their link properties, and rules governing how types are affected by links:

- Figure 8 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.
- Figure 9 lists the input/output types of six components, whose input/output types are listed in , along with the types produced by the source, *bbc.com*, and that required by the sink, MediaPlayer. Of the six, the *splitter* and the *padder* are Windows Media SDK based services; the first splits a video+audio ASF stream into an audio-only ASF stream available via HTTP, while the second "fills in" legal media frames whenever its input stream stops. The other four components are drivers, which cooperate

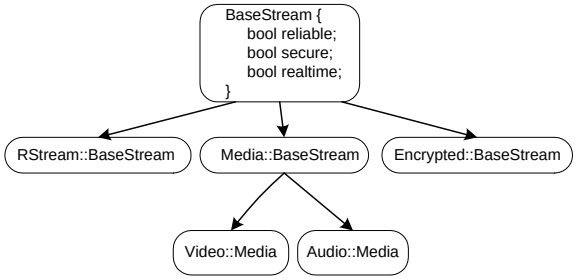


Figure 8. Hierarchical type definition of case study.

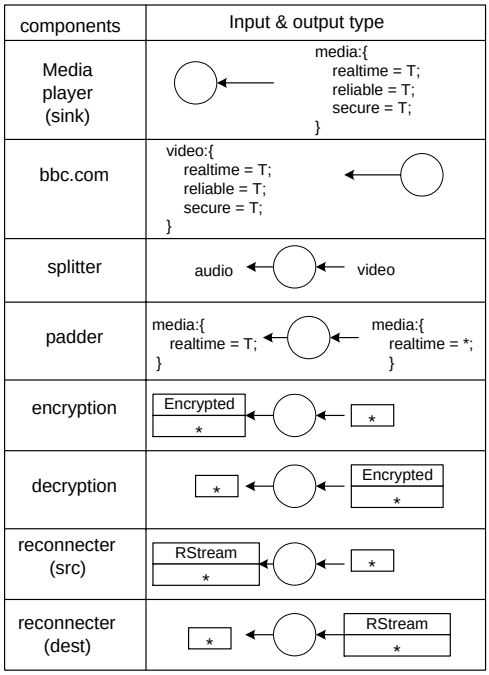


Figure 9. Input and output of case study components.

to handle encryption (*encryption* and *decryption*) and reliable transmission (*reconnecter(src)* and *reconnecter(dest)*) respectively.

- Table 1 shows the properties of the wired and wireless links and Table 2 shows how these effect different types. In Table 2, "effect isolation" refers to a type isolating the effect of a link property for data type instances below it in the type stack.

Figure 7 also shows the deployed components for each of the stages. For Stage 1, whose deployment is triggered when the application issues a connect call to an external streaming service, the plan consists of the *padder* and *reconnecter(dest)* running on the laptop, and *reconnecter(src)* running on the intermediate EE. These components are selected so as to guarantee the real-time and reliability requirements of the type expected at the sink. Not shown are communication adapters required for hooking up the application with the EE, and the EE

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

Table 1. The properties of links used in case study.

	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

Table 2. The effects of link properties on data types.

with the server, *bbc.com*. Needless to say, MediaPlayer receives a continuous data stream via the CANS components and is able to render it without any problems.

Stage 2 starts when we disconnect the laptop from the wired LAN. The *padder* ensures that MediaPlayer continues to receive legal frames even when no data is coming from the network. The communication adapters running on the laptop and the intermediate host detect the disconnection and reconnect to each other using the wireless interface, with data continuity at the semantic segment level ensured because of the *reconnecter(src)* and *reconnecter(dest)* drivers. At this time, because the wireless link has the *secure* property set to false, *encryption* and *decryption* drivers are installed into the data path automatically. Note that this adaptation involves the data path reconfiguration algorithm described in Section 4.2 to flush any in-transit segments.

Stage 3 starts when as the laptop is moved away from its access point, bandwidth drops below a threshold. The event detecting this triggers deployment of a new plan, resulting in the instantiation of the *splitter* component, capable of reducing stream bandwidth requirements. However, *splitter* supplies data of type `Audio`, which while compatible with the type specifications, `Media`, of the client application and other deployed components, requires the application to be placed in a different mode. Achieving the latter requires us to go outside the CANS infrastructure. Currently, we set up an ASX file that forces MediaPlayer to reconnect when the first connection is shutdown. This reconnect request is trapped and used to initiate the new plan. While this works, it also points out the need for a better abstraction of the protocol between applications and the infrastructure. Recent work by Lara et al. [4] of application adaptation relying on component automation interfaces points to what might be a promising direction.

Overall, the case study successfully demonstrated all of the important features supported by the CANS infrastructure: dynamic type-based composition and planning,

event-driven adaptation that spans multiple drivers, and integration of legacy applications and services.

6 Related Work and Discussion

CANS shares its goals with many recent efforts that have looked at injecting adaptation functionality into the network. Instead of describing each separately, we group related efforts to put our work in perspective.

Adaptation functionality can be introduced only at the end-points or could be distributed on intermediate nodes. Odyssey [15], Rover [12] and InfoPyramid [14] 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. Efforts targeting adaptation at intermediate nodes in the network can themselves be viewed in terms of two issues: whether adaptation functionality is application-transparent or application-aware, and whether the functionality is introduced at the network level or the application level.

Systems such as transformer tunnels [16], protocol boosters [13] are examples of application-transparent adaptation efforts that work at the network level. Such systems can cope with localized changes in network conditions but cannot adapt to behaviors that differ widely from the norm. Moreover, their transparency hinders composability of multiple adaptations. More general are programmable network infrastructures, such as COMET [3], which supports flow-based adaptation, and Active Networks [17, 19], which permit special code to be executed for each packet at each visited network element. While these approaches provide an extremely general adaptation mechanism, significant change to existing infrastructure is required for their deployment.

Similar functionality can also be supported at the application level. The cluster-based proxies in BARWAN/Daedalus [6], TACC [7], and MultiSpace [9] 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 [20], 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. More general are systems such as Ninja [8], PIMA [2], and Portolano [5], which permit construction of programmable ubiquitous access systems from networked services and transformational components. CANS also provides application-level support for injecting application-aware functionality into the network, but differs from the above systems in its focus on infrastructural support required for dynamic adaptation.

CANS has been most heavily influenced by the Conductor design and shares several features with the Ninja infrastructure. Conductor [20] provides an application-transparent adaptation framework that permits the introduction of arbitrary adaptors in the data path between applications and end services. Applications are integrated into the framework by modifying the kernel to trap calls that create and use TCP sockets. CANS borrows the idea of transparent stream-based adaptation from Conductor but differs in applying it to application-aware adaptation in a larger context that involves multiple services contributing to the data path; consequently, we require infrastructural support for downloading component code, instantiating the components, and ensuring compatibility. Also different is the degree of support provided by the infrastructure for reconfiguring existing paths, specifically the notion of semantics-preserving adaptation that spans multiple drivers, and general support for dynamic run-time composition of components.

Ninja [8] is a general architecture for building robust internet-scale systems and services consisting of three components: services, units, and paths. We restrict our attention to how paths are constructed in Ninja since that is the closest to our objective. Several CANS concepts find close matches in the Ninja design: our service-driver distinction is closely related to Ninja's service-operator distinction and both systems share ideas such as type-based composition and dynamic service adaptation. Despite these high-level similarities, the systems differ significantly in the details. Unlike Ninja, the CANS infrastructure provides support for (1) efficient composition of multiple drivers within the same physical host, (2) planning algorithms that consider route characteristics in addition to bridging type incompatibilities, (3) dynamic and distributed event-driven adaptation on existing paths, and (4) support for semantics-preserving adaptations that span multiple drivers; Ninja requires applications to provide their own mechanisms to ensure semantics such as guaranteed or in-order data delivery. On the flip side, it must be noted that unlike Ninja, CANS currently provides little support for scalability.

7 Conclusions

This paper has presented an application-level infrastructure, CANS, for injecting application-specific functionality into the data path connecting applications and end services. Such functionality can monitor and adapt to resource changes, providing the basic support needed for building novel application-level services that can seamlessly integrate diverse end devices across heterogeneous networks. Main contributions of CANS include: (a) efficient dynamic composition of components by requiring they adhere to a restricted interface, (b) dynamic and distributed adaptation using distributed events and novel path reconfiguration algorithms, and (c) support for legacy applications and services. Our experience indicates that CANS conveniently permits dynamic deployment and distributed adaptation of application-aware components to improve user experience.

CANS is one component of a larger project, Computing Communities, which focuses on distribution middleware for legacy applications. Our future work involves integrating CANS with related efforts emphasizing resource management and security issues, improving its performance, and designing better planning algorithms.

Acknowledgments

We thank the anonymous USITS reviewers for helping us improve this paper. This research was sponsored by DARPA agreements F30602-99-1-0157 and N66001-00-1-8920; by NSF grants CAREER:CCR-9876128 and CCR-9988176; and Microsoft. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies or endorsements, either expressed or implied, of DARPA, Rome Labs, SPAWAR SYSCEN, or the U.S. Government.

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] G. Banavar and et al. Challenges:an application model for pervasive computing. In *Proc. of the Sixth ACM/IEEE Intl. Conf. on Mobile Networking and Computing*, August 2000.

[3] A. T. Campbell and et al. A survey of programmable networks. *ACM SIGCOMM Computer Communication Review*, April 1999.

[4] E. de Lara, D. S. Wallach, and W. Zwaenepoel. Puppeteer: Component-based adaptation for mobile computing. Technical Report TR-00-360, Computer Science Department, Rice University, October 2000.

[5] M. Esler, J. Hightower, T. Anderson, and G. Borriello. Next century challenges:data-centric networking for invisible computing. the portolano project at the university of washington. In *Proc. of the Fifth ACM/IEEE Intl. Conf. on Mobile Networking and Computing*, August 1999.

[6] 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.

[7] 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.

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

[9] 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.

[10] J. Haartsen. Bluetooth– the universal radio interface for ad hoc, wireless connectivitity. *Ericsson Review*, 1998.

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

[12] 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.

[13] A. Mallet, J. Chung, and J. Smith. Operating system support for protocol boosters. In *Proc. of HIPPARCH Workshop*, June 1997.

[14] 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.

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

[16] P. Sudame and B. Badrinath. Transformer tunnels: A framework for providing route-specific adaptations. In *Proc. of the USENIX Technical Conf.*, June 1998.

[17] D. Tennenhouse and D. Wetherall. Towards an active network architecture. *Computer Communications Review*, April 1996.

[18] U. Varshney and R. Vetter. Emerging mobile and wireless networks. *Communications of the ACM*, pages 73–81, June 2000.

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

[20] 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.