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

The growth of the internet has been fueled by an increasing number of sophisticated network-accessible services. Unfortunately, the high bandwidth and processing requirements of such services is at odds with current trends towards increased variation in network characteristics and a large diversity in end devices. Ubiquitous access to such services requires the injection of additional functionality into the network to handle protocol conversion, data transcoding, and in general bridge disparate portions of the physical network. Several researchers have proposed infrastructures for injecting such functionality; however, many challenges remain before these infrastructures 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) dynamic and distributed adaptation of injected components in response to system conditions; and (c) support for legacy applications and services. The network view supported by CANS consists of *applications*, stateful *services*, and *data paths* between them built up 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 run-time type-based composition model dictates how new services and drivers are integrated with existing components. An interception layer that virtualizes network bindings permits legacy applications to plug into the CANS infrastructure, and a delegation model does the same for legacy services.

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

1 Introduction

The emergence of new networking technologies such as broadband to the home, Wireless 3G [17], and Bluetooth [9], coupled with increasing numbers of network-capable communication and computation 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 components are able to detect this, activate the wireless card on her laptop, and seamlessly switch over data transmission to a local wireless network. Recognizing that the bandwidth on the wireless LAN is insufficient to continue delivering continuous video at the original resolution and rate, the components automatically downgrade picture quality. Shortly after this, Alice has to leave 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 components further downgrade the media stream (say to only include the audio part), while recording the full stream at a server that Alice can check offline.

Although the above is a compelling scenario, it imposes several requirements that are inadequately handled by current-day internet infrastructure: rapid creation and deployment of new services, mobility, application-aware computation in the network, and dynamic and distributed adaptation. Even if such services can be created, the existing view which hides network characteristics from the application and treats services as isolated islands is at odds with the large variation in network and end-device characteristics. Current-day data paths between communicating parties can include links with very different bandwidth, delay, and error characteristics, ranging from serial links to wireless to broadband to fiber backbones. Hiding these differences from the application will result in unsatisfactory application performance, and the alternative of providing different levels of service access 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 monitor changes in resource characteristics of end-devices and network links and can dynamically adapt to them by handling activities such as protocol conversion, data transcoding, etc. Several researchers have proposed infrastructures for injecting such functionality, ranging from end-point solutions [11, 14] to more distributed solutions that introduce application-aware functionality at intermediate sites either at the network level [16, 3, 19] or at the application level [1, 5, 8]. Although these systems have articulated the high-level architectural requirements of such solutions, many challenges, particularly with respect to dynamic application services management and composition, remain before these infrastructures see widespread deployment.

This paper presents the architecture and implementation of **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*: The infrastructure must permit separately defined application-aware components to be dynamically instantiated and connected to each other; these components should interact using efficient mechanisms where possible (e.g., shared memory within a host).
- *Dynamic and Distributed Adaptation*: The infrastructure must permit dynamic adaptation to environment changes; such adaptations must incur low overhead and maintain overall application semantics.
- *Support for Legacy Applications and Services*: In addition to providing a simple interface for new applications, legacy applications and services should be integrated into the infrastructure 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 model assists in distributed adaptation, and a run-time 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. An interception layer that transparently virtualizes network bindings, currently TCP sockets, permits legacy applications to plug into the CANS infrastructure, and a delegation model does the same for legacy services.

Our current implementation of CANS works with Windows NT/2000 clients and Java/RMI-capable intermediate hosts. The clients and hosts run the CANS execution environment, which leverages Java language support to dynamically create, migrate, and adapt drivers and services. Early experience with a case study involving a shrink-wrapped client application (Windows Media Player) in a dynamically changing network environment indicates that our approach is promising: the flexible mechanisms in CANS permit dynamic deployment and distributed adaptation of application-aware components to improve overall user experience.

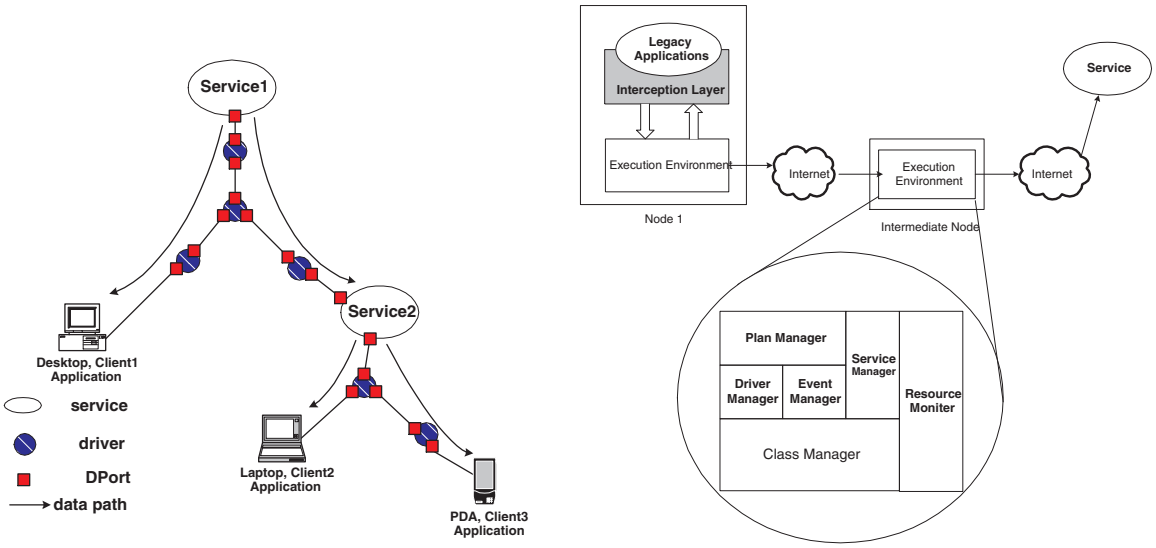
The rest of this paper is organized as follows. Section 2 presents the CANS architecture. The individual components and support for distributed adaptation are described in more detail in Sections 3 and 4. Section 5

presents our current implementation of CANS and describes our experience with using it in the context of a case study. Section 6 discusses related efforts to place our work in context, 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. Traditionally, the functionality of a data path is restricted to transmitting data between the end points. The CANS infrastructure extends this notion to enable end services, applications, or some other entity to dynamically inject application-specific components into the network; these components customize the data path with respect to the characteristics of the underlying physical network links and properties of the end device as well as dynamically adapt to any changes in these characteristics (see Figure 1(a)).



(a) (b)

Figure 1: (a) Basic organization of CANS, (b) Interaction between legacy applications and CANS architecture.

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). For example, the data from an HTTP server has type HTTP and can be consumed by any component that understands this type. Injected components come in two flavors: stateful *intermediate services* (referred to as just *services* in the rest of the paper) 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.

Legacy applications interface with the injected components using an *interception layer* (see Figure 1(b)) that transparently virtualizes the network bindings of the application, in our case TCP sockets. Logically, data to and from the application using a particular socket is sent via multiple CANS components, while retaining the illusion from the application's perspective of an end-to-end TCP connection. Legacy services are just as easily integrated; a *delegate object* controls and represents a service in its interactions with the CANS infrastructure.

The CANS network is created dynamically, based upon various parameters such as user, client, and

service information, as well as characteristics of the underlying platform. The components which constitute a data path, the composition order between 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.

To show an example use of the CANS infrastructure, consider a hand-held device with a WML-capable web browser. When a user requests a page, the CANS interception layer traps the request and upon realizing that (1) there is a proxy service running in the local domain, (2) that the requested page contains HTML data, and (3) there exists a driver capable of transcoding HTML to WML, dynamically creates a data path between the application and the proxy server that includes the HTML-to-WML driver.

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 run-time 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 for creating the initial plan comprising drivers, services, and data paths in response to a request trapped by the interception layer, and replanning in response to change in system conditions.

The driver and service manager maintains information about deployed drivers and manages data path operations, including inserting new drivers, creating new services, and reconfiguring existing paths as required. Driver connections that span EEs are multiplexed onto the inter-EE connection.

The event manager is responsible for receiving both system-level and component-level events and propagating these on to interested components. Section 4 describes the event model in more detail.

The resource monitor monitors system conditions, informing the event manager when registered trigger conditions fire. For example, events can indicate changes in network interface state, CPU availability, etc.

3 CANS Components

In this section, we describe the core CANS components—drivers and services—and auxillary components required to connect execution environments with each other and with applications and legacy services.

3.1 Drivers

Drivers serve as the basic building block for constructing adaptation-capable, customized data paths between applications and services. Drivers are just stand alone 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. Drivers satisfy the following properties:

1. Drivers receive and send data using a standard *data port* interface, called a `DPort` (see Figure 2). `DPorts` are distinguished based on whether they are being used for input into the driver or output from the driver. Each `DPort` has associated with it data type information, which influences how drivers are composed with each other. The primary responsibility of a driver is to process input typed data and create output typed data on its ports.
2. Drivers are *passive* objects, moving data from input ports to output ports in a purely demand-driven fashion. Thus, driver operations are triggered either when one of its output ports is checked for data, or one of its input ports receives data.

3. Drivers consume and produce data in the granularity of an integral number of application-specific units, called *semantic segments*. Informally, this ensures that an input segment is either completely or not at all reflected in output segments produced by the driver. For example, a driver converting HTML data into WML must wait for a complete HTML unit to be received before it sends out the corresponding WML unit. Semantic segments are the units of transmission (and any required retransmission), and permit the execution environment to infer which output segments can be affected by the contents of a particular input segment.
4. Drivers contain only *soft state*, which can be easily 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.

The first two properties enable dynamic composition (see below) and efficient transfer of data segments between multiple drivers that are mapped to the same physical host. Specifically, the run-time system can employ a single thread 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 adaptation, either within a single driver or across data path segments while preserving application semantics; Section 4 discusses the path reconfiguration algorithm in more detail.

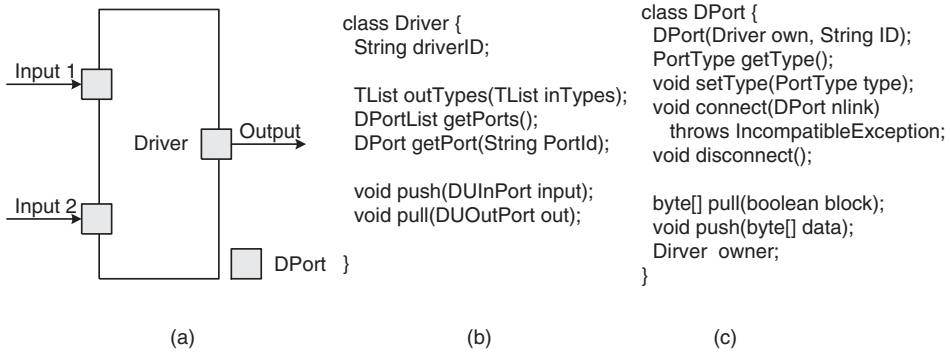


Figure 2: Definition of driver(a) Driver, (b) Driver interface, and (c) DPort interface.

Type-based Composition

The composability of CANS components (both drivers and services) is decided by compatibility of the data types associated with the input and output ports being connected. Essentially, an output `DPort` of driver D_1 is disallowed from connecting with the input `DPort` of driver D_2 unless the data type associated with that output port is compatible with the required data type of the input `DPort`. CANS data types can be hierarchical, similar to the notion of type hierarchies in object-oriented languages. More specific types can be substituted wherever a generic type is required. For example, if data type C_1 is a subclass of data type C_2, it means C_1 can be used wherever data type C_2 is needed.

However, instead of defining constant data types for their output ports, CANS components maintain *run-time type information* about the data produced on their outputs. Each component defines a function mapping its input data types into output data 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 data type set for the ith input port, and T_{out_j} is the resulting data type produced on the jth output port.

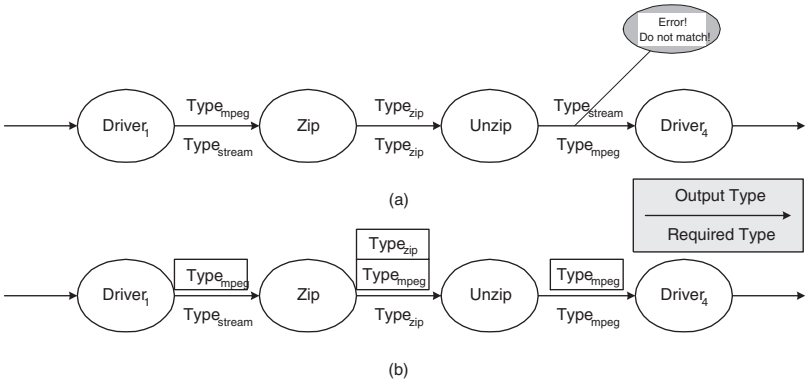


Figure 3: A simple example of type compatibility: (a) flat scheme, (b) stack scheme.

Maintenance of run-time type information is required to avoid type information loss during composition, which can prevent composition of valid components. Figure 3(a) shows an example of such behavior. The input type required by the **Zip** driver is Type_{stream}, and it produces Type_{zip}. The input type required by the Unzip driver is Type_{zip}, and its output type is Type_{stream}. Suppose the output of $\mathbf{Driver}_1$ is Type_{mpeg} (a subtype of Type_{stream}) while the input type required by $\mathbf{Driver}_4$ is Type_{mpeg}. Although such a composition seems perfectly legal, the output of the **Unzip** driver is calculated to be of type Type_{stream}, which is not compatible with the input requirements of $\mathbf{Driver}_4$. The culprit here is that information about the data being of type Type_{mpeg} is not available at the output port of the Zip driver.[1] To solve this problem, we maintain port data types in a stacked form, using a structure called `PortType`. The top of the `PortType` stack is the current data type associated with that port (`PortType(`p`).top` for `DPort` p). The operations on `PortType` includes *push, pop, peek,* and *clone*, which have the standard meanings. The type compatibility between an input and an output port is now determined by checking the top of the `PortType` stack against the required type. Type information flows downstream automatically at run time when two `DPorts` get connected. As demonstrated in Figure 3(b), with this information, the earlier composition is now admissible.

3.2 Services

The second core CANS component are *services*. Services are standalone, relatively heavyweight execution entities that supply applications with data streams of a particular type. They have all of the requirements of services running in conventional network infrastructures: high availability, robustness, etc. Unlike drivers, CANS imposes relatively few interface requirements on services. A service is required to register itself with the CANS infrastructure identifying the data types it supports, and optionally providing a *delegate object* that can control the service and act on its behalf in interactions with the rest of the CANS infrastructure. The delegate object implements a standard (small) interface consisting of activating and suspending the service, and receiving CANS events. Services can export data using either the `DPort` interface described earlier, or using any standard internet protocol such as TCP, HTTP, etc.

The CANS infrastructure provides applications with a general platform to create, compose, and control services across the network. Service composition follows the same algorithm as driver composition, using the identified types supplied at registration time.

The primary distinction between drivers and services is the fact that the former represents rigidly con-

[1] This information can be inferred by examining the entire data path, but this precludes local decision making.

strained, mobile, soft-state adaptation functionality, while the latter can 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 services: 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 service migration is not explicitly supported by the CANS infrastructure; CANS can create new instances of the service on nodes capable of hosting the service, but it is upto the running service as to how it manages state transfer. This design choice reflects the view that services are migrated infrequently and doing so requires service-specific protocols that are difficult to abstract cleanly.

3.3 Communication Adaptors

Communication adaptors are auxillary CANS components, which transmit data *physically* across the network. These components supply data to CANS drivers and consume data produced by them. To do this, they expose the same `DPort` interface, appearing to other drivers just as a regular driver. However, the role they play is to implement three kinds of logical connections: (1) between applications and drivers; (2) between two drivers mapped to different execution environments; and (3) between a driver and a service that exports data using an interface other than `DPort`.

To provide this functionality, communication adaptors 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 take advantage of existing mechanisms (e.g., different network interface adapters, and different transport protocols) best matched to the characteristics of the underlying network. Thus, a logical connection between two drivers that are mapped to different execution environments is realized using a pair of communication adaptors, one on each of the involved hosts. The drivers are connected to the adaptors on the local host, and the adaptors establish a physical communication link between themselves.

Communication adaptors can encapsulate behaviors that permit them to adapt to and recover from minor variations in network characteristics. For instance, communication adaptors can be written to use one of several network alternatives, automatically transitioning between them to improve performance. For example, an adaptor on a laptop with both wireless and wired network connections can automatically start using the wireless connection when it detects disconnect of the wired interface. The continuity semantics on such reconnection are dictated by the requirements of the data types associated with the adaptor'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. Currently, we focus on TCP sockets, but the support could just as easily be extended to other protocols. CANS components obtain data from and send data to these virtualized sockets, with the application retaining the illusion that it is interacting with the network. The interception layer is injected using a technique known as API interception [10], which relies on a run-time rewrite of portions of the memory image of the application (either the import table for functions in dynamically-linked libraries (DLLs) or the headers of arbitrary functions).

4 Distributed Adaptation in CANS

The CANS infrastructure supports three modes of adaptation in response to dynamic changes in system components: (1) *intra-component adaptation*, where each service or driver detects and adapts to minor resource variations on its own; (2) *data path reconfiguration*, where the data path undergoes localized changes involving the 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 variations in system conditions. 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 driver and service can incorporate its own adaptation behavior, which may or may not be coordinated with adaptation in other CANS 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 isolated from that in another as long as the effect of adaptation is restricted to be within a single semantic segment (see Section 3.1). To simplify the expression of such adaptation behaviors, the CANS infrastructure provides distributed event propagation support, permitting components to raise arbitrary events as well as listen for specific ones. This support can also be used to trigger coordinated adaptation, where one component raises an event that is listened to by another.

The kernel of the CANS event model is the **Event Manager**, one per execution environment. The Event Manager is responsible for catching events, firing events, and transmitting events across networks. Any component in the system (including delegate objects for legacy services) can register itself as a listener of specific events from specific event sources, and it can also raise an event on the Event Manager, which in turn fires it to corresponding listeners. 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 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 that can trigger replanning.

4.2 Data Path Reconfiguration using Semantic Segments

Data path reconfiguration involves insertion, deletion, or reordering of drivers along an active data path. While this provides great flexibility in responding to a range of resource variations, the fundamental problem encountered 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 data path segment that is being affected by the reconfiguration can have some stream data that is partially processed; either in the internal state of drivers, or in transit between execution environments. 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 data loss or in-order reception.

Figure 4 shows an example highlighting this problem. Driver d_0 is the 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 the renderer of the MPEG data. d_1 and d_2 are drivers that need to be removed. When driver d_0 has output 2 frames, d_3 receives one frame, but with the system in this state, d_1 and d_2 cannot be removed from the data path. The reason is that doing so affects semantic continuity, since it is incorrect to retransmit either of

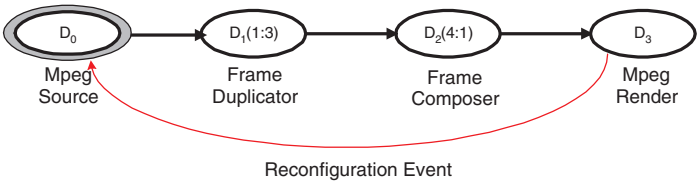


Figure 4: An example of data path reconfiguration using semantics segments.

the second segment from d_0 (whose affects have been partially observed at d_3), or the third segment (which would result in a loss of continuity at d_3).

CANS provides an efficient solution to the data path reconfiguration problem, which does preserve semantic continuity. Our solution leverages restrictions placed on driver functionality, specifically the soft-state and semantic segments assumptions described in Section 3.1. Our scheme relies on a notion we call *delayed operation*, which defers the reconfiguration to a point where the system can guarantee continuity and exactly once semantics.

In our description here, we restrict our attention to reconfiguring a portion of the data path that has only one input and one output. Internally, it is possible for the portion to have an arbitrary graph of drivers. The basic idea is that reconfiguration is delayed until the driver immediately downstream of the portion being reconfigured (d_3 in our example) *completely receives the effects of every segment* sent out by the driver immediately upstream of the portion (d_0 in our case) prior to the request for reconfiguration. Note that in order to achieve this, we need to continue transmitting from the upstream driver, in effect flushing out the contents of the intermediate drivers. Once the downstream driver indicates, using the distributed event system, that it has received the last of the affected segments, the system can remove the portion of the data path, replacing it with a compatible set of new drivers. To restart the system, the upstream driver retransmits starting from the first segment that was not completely received. The fact that the system needs to wait for only a finite number of segments to flow downstream follows from the semantic segments restriction we have placed on our drivers. This number, can in fact, be computed based entirely on the properties of drivers making up the portion that is being reconfigured simply by tracking which input segments produce which output segments.

For our example, the CANS reconfiguration algorithm works as follows (reconfiguration is requested after segment 2 has been sent out by d_0):

1. CANS marks d_0 as the upstream point and d_3 as the downstream point, which causes d_0 to buffer every segment it sends out after this time. Except for buffering, d_0 keeps sending out data as usual.
2. When downstream point d_3 realizes that it has received a complete segment from the upstream point, it freezes its input port and raises an event to the plan manager. In this case, this happens when segment number 3 is received, which corresponds to segment number 4 input by d_0 (this follows from the fact that the aggregate effect of d_1 and d_2 is to produce $3/4$ the number of segments they see on their input).
3. The plan manager can now freeze d_0, remove d_1 and d_2, and replace them 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.

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. As described earlier, planning is triggered when the interception layer detects a connect attempt on a TCP socket of interest. The key component responsible for planning

in the CANS infrastructure is the *plan manager*. The plan manager takes responsibility both for creating the original plan, as well as changing it as required based on evolving system conditions. Replanning is triggered by events from the resource monitor that are propagated to the *plan event delegate* by the event manager.

All of the components on a plan must satisfy the type compatibility requirement and the components at the end points must compatibly interface with the application and end service. To generate a strictly optimal plan is a challenging problem because the candidate space could grow exponentially with the number of compatible components. Our current solution for this is to use type compatibility and an application-specific adaptation policy to guide selection among candidate plans, using simple heuristics to find a reasonable one. The heuristic that we use currently consists of two steps. The first step selects a *route*, spanning multiple network domains, where the hosts are chosen based upon service availability requirements. Route selection attempts to maximize the minimum bandwidth available along the route. The second step *partitions a driver sequence* among these hosts to construct data paths between services. The selection of which drivers form the data path and where they reside is handled by an integrated heuristic that takes into consideration the *bandwidth effect factor* of each driver and seeks a data path with the least bandwidth bottleneck. To reduce the search space, we start with a feasible data path and then try to reduce the bottleneck factors by inserting/removing components.

The application-specific policy plays an important role in plan selection. For example, if the application specifies that audio data is acceptable if the video frame rate is lower than 10 frames/sec, the plan manager can add an audio/video splitter at some point upon detecting that the bottleneck bandwidth falls below this threshold.

5 Experience with Using CANS

We have been experimenting with a prototype implementation of CANS architecture. The current focus has been more on evaluating the suitability of our approach for injecting application-specific functionality into legacy client-server network data paths, as opposed to developing the highest performing implementation.

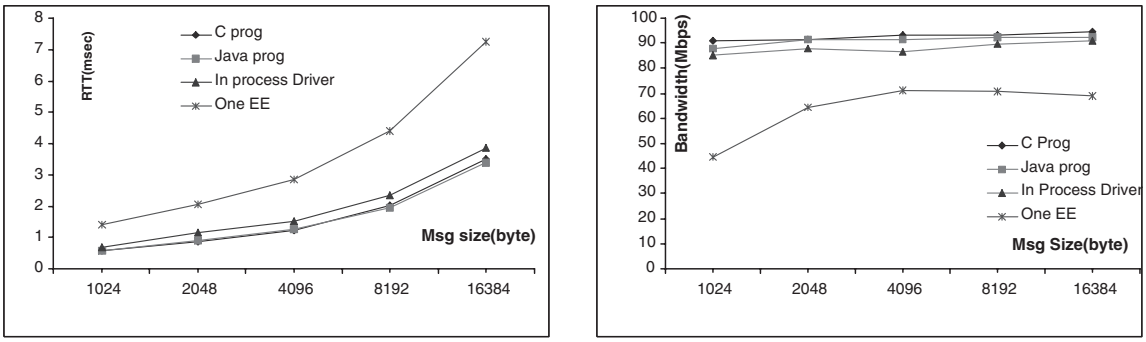
The current implementation of the CANS infrastructure works with Windows NT/2000 clients and Java/RMI-capable intermediate hosts. Both the execution environment (EE), which runs as a user-level process within a JVM, and driver components are written in Java. Java was chosen because of its support for code mobility, safety, and strong typing. The interception layer for integrating legacy applications is written as a native DLL, which is injected into the application at start time, and reroutes application socket calls by rewriting the memory code image of the application. We make use of the Detours toolkit under Windows NT/2000 [10] that provides DLL injection and function rerouting facilities. The interaction between wrapper and EE is based on the standard JNI interface.

To set up a customized data path for an application, the interception layer first obtains the address of an *agent node*, permitting it to interact with the EE running there. The plan manager on that node builds the plan, instantiates it, partitions it, and then downloads plan fragments to individual environments. Interactions between different EEs, and between the application and its agent node make use of Java/RMI. Data transmissions between components, which are more performance critical, makes use of the communication adaptors described earlier.

In the rest of this section, we first describe some microbenchmarks that capture the 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 Intel Pentium II 450Mhz nodes, with 128 MB main memory, running Windows NT 4.0, and connected using 100 Mbps switched Ethernet.



(a) Round Trip Time (b) Bandwidth

Figure 5: Overhead of the CANS infrastructure, measured in terms of latency and bandwidth impact on communication between an application and an end service.

Figure 5 shows the overheads introduced by the CANS infrastructure, measured in terms of their impact on the 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 interception layer running within the application process and indicates the basic overheads of driver composition, and the latter considers the case where the data path includes null drivers sitting on an intermediate host between the application and service.

Figure 5 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 additional 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 the CANS architecture provides enough flexibility to support large-scale applications, we used the prototype in a case study involving a shrink-wrapped application: Microsoft MediaPlayer. The goal of the study was to see if CANS can be used to make the application adapt to a dynamically changing network environment, *without* embedding such functionality into the application itself. For our case study, we implemented a service (using Windows Media SDK) which obtains ASF data from the original server and can optionally splits it (under external control) into audio and video components available using HTTP on different ports.

In addition to this service, other CANS components included Null (forwarding), Zip and Unzip drivers, and communication adaptors that could (1) provide data on the application's virtualized sockets; (2) transmit data between EEs; and (3) interact with a service using the TCP protocol. The first and second adaptors

were capable of supporting using multiple network interfaces, automatically transitioning between active interfaces in a fixed priority.

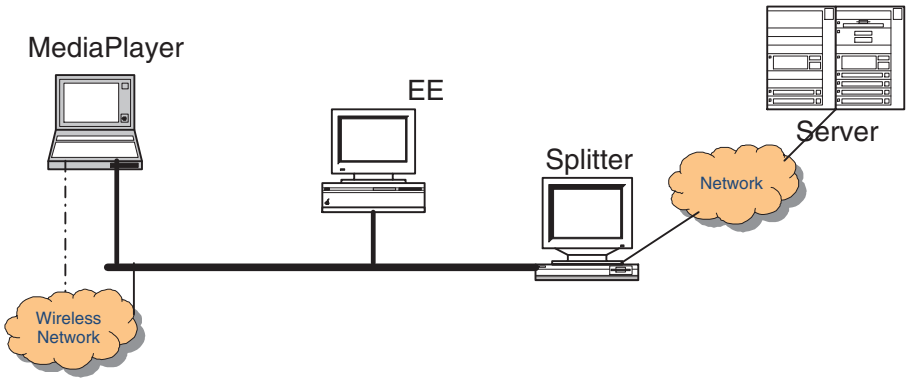


Figure 6: Case study: MediaPlayer with CANS infrastructure.

As showed in Figure 6, the experimental environment consisted of a laptop which has both wireless and wired network interfaces, a desktop which can run the splitter service, and an intermediate computer capable of hosting an execution environment, which also acts as the agent node for applications running on the laptop. The experiment progresses in four stages described below:

To start the experiment, we started a wrapper process that injected the sockets interception layer into the MediaPlayer application. When the application issues a connect call to an external streaming service, the interception layer intercepts the call and forwards the request to the agent node running on the intermediate host. The plan manager there computes the (hardcoded) initial plan, instantiates the necessary components, and makes the connections. The deployed plan consists of the splitter service, forwarding drivers running on the intermediate EE, and communication adaptors hooking up the application with the EE, and the EE with the service. Needless to say, MediaPlayer receives a continuous data stream via the CANS components and is able to render it without any problems.

The second stage of the experiment starts when we disconnect the wired connection between the application node and the rest of the system. The communication adaptor running there detects this and reconnects using the wireless interface, while preserving data continuity on its output ports hooked into the application's virtualized sockets. This demonstrates the first of CANS' adaptation modes: intra-component adaptation.

The third stage of the experiment involved moving the laptop away from its access point, resulting in a bandwidth drop. The resource monitor running in the application EE detects this (by comparing stream bandwidth against a preset threshold) and triggers an event that is caught by the plan manager. The plan manager responds to this (again using a hardcoded plan) by reconfiguring the data path between the application node and the intermediate EE node to include a Zip and an Unzip driver.[2] Note that this adaptation involves the data path reconfiguration algorithm described earlier to flush any in-transit segments between the null drivers on the intermediate EE and the communication adaptor on the application node.

The fourth stage of the experiment involved moving the laptop further away from the access point, triggering another event from the resource monitor and additional replanning. The plan corresponding to this event requires the service to create a new port supplying only audio output, and deploying a new plan to supply the application with this differently formatted input. To implement this, the infrastructure needs to propagate an event to the splitter service, and force the application to reconfigure itself to accept audio where it was previously receiving video. The first was naturally handled using a service delegate object

[2]Adding Zip/Unzip to the already compressed media stream has little effect; here we use these drivers as placeholders for more application-specific transformers (e.g., those that work by dropping frames).

described earlier; however, the second required us to go outside the CANS infrastructure. We set up an ASX file that forced MediaPlayer to reconnect when the first connection was shutdown. This reconnect request was trapped and used to initiate the new plan. While this experiment demonstrated the effectiveness of the CANS infrastructure in supporting global adaptation involving changes to multiple components, it also pointed out the need for a better abstraction of the protocol between applications and the infrastructure.

Overall, the case study successfully demonstrated all of the important features supported by the CANS infrastructure. First, starting only from a specification of the data type required by MediaPlayer, CANS was able to compose services and drivers (type compatibility was checked at run time) and deploy the connection plan. Second, distributed event propagation and all three of the adaptation modes of CANS were highlighted showing that CANS provides enough flexibility to implement them. We did run into a problem because of our choice of MediaPlayer as the client application: MediaPlayer would detect when it was not receiving data at a fast enough rate on a connection (which would happen whenever CANS was adapting the data path), close this connection, and try and reconnect. The ideal solution for this would be to have a MediaPlayer-specific driver that would supply the application with null legal frames; however, this requires knowledge of the media format which is proprietary. Our solution here (admittedly crude) was to freeze the application when we were adapting the data path. This problem only reinforces the need for an infrastructure that is application-aware (such as CANS) instead of trying to adapt completely transparently.

6 Related Work

CANS shares its goals with many recent efforts that have looked into 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 [14], Rover [11] and InfoPyramid [13] 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 [15], protocol boosters [12] 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 [16, 18], 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 [5], TACC [6], and MultiSpace [7] 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 [19], 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 true because of non-technical reasons such as those we encountered in our case study. More general are

systems such as Ninja [8], PIMA [2], and Portolano [4], 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 [19] 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 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) event propagation and distributed adaptation across multiple intermediate hosts, and (3) 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 explicit support for scalability and fault recovery.

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 and cope with variations in network characteristics. The main contributions of CANS include: (a) efficient and dynamic composition of injected components by requiring them to adhere to a restricted interface, (b) dynamic and distributed adaptation that takes advantage of distributed event propagation and novel path reconfiguration algorithms, and (c) support for legacy applications and services. Early experience indicates that our approach is promising, and that the flexible mechanisms in CANS permit dynamic deployment and distributed adaptation of application-aware components to improve user experience.

CANS is one component of a larger project at New York University, Computing Communities, which is looking at developing transparent distribution middleware for legacy applications. Our future work involves integrating CANS with efforts that are looking at resource management, security, and reliability issues, improving the performance of the infrastructure, particularly with respect to synchronization and data copying overheads, and designing more sophisticated planning algorithms.

References

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

[2] G. Banavar, J. Beck, E. Gluzberg, J. Munson, J. Sussman, and D. Zukowski. Challenges:an application model for pervasive computing. In *Proceedings of the Sixth ACM/IEEE International Conference on Mobile Networking and Computing*, pages 266–274, August 2000.

[3] A. T. Campbell, H. G. De Meer, M. E. Kounavis, K. Miki, J. Vicente, and D. Villela. A survey of programmable networks. *ACM SIGCOMM Computer Communication Review*, April 1999.

[4] 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 *Proceedings of the Fifth ACM/IEEE International Conference on Mobile Networking and Computing*, pages 256–262, August 1999.

[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 *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997.

[7] S. D. Gribble, M. Welsh, E.A.Brewer, and D. Culler. The multispace:an evolutionary platform for infrastructual services. In *Proceedings of the 1999 Usenix Annual Technical Conference*, June 1999.

[8] S. D. Gribble, M. Welsh, R. von Behren, E. A. Brewer, D. Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R.H. Katz, Z.M. Mao, S. Ross, and B. Zhao. The ninja architecture for robust internet-scale systems and services. *Special Issue of IEEE Computer Networks on Pervasive Computing*, 2000.

[9] J. Haartsen. Bluetooth– the universal radio interface for ad hoc, wireless connectivitity. *Ericsson Review*, (3), 1998.

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

[11] 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.

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

[13] 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.

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

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

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

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

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

[19] M. Yavis, A. Wang, A. Rudenko, P. Reiher, and G. J. Popek. Conductor:distributed adaptation for complex networks. In *Proceedings of the seventh workshop on Hot Topics in Operating Systems*, March 1999.