Panda: middleware to provide the benefits of active networks to legacy applications — Source link

V. Ferreria, Alexey Rudenko, Kevin Francis Eustice, Richard Guy ...+2 more authors

Institutions: University of California, Los Angeles

Published on: 29 May 2002

Topics: Middleware (distributed applications) and Legacy system

Panda: Middleware to Provide the Benefits of Active Networks to Legacy Applications[1]

Vincent Ferreria, Alexey Rudenko, Kevin Eustice, Richard Guy, V. Ramakrishna, and Peter Reiher
UCLA

Abstract

Panda is middleware designed to bring the benefits of active networks to applications not written with active networks in mind. This paper describes the architecture and implementation of Panda, and provides data on the overheads incurred and performance benefits achieved. The paper also discusses some of the key issues of automatically and transparently intercepting data streams and converting them into active streams, including interception mechanisms, automated planning facilities, and allowing user and application control of the middleware.

1. Introduction

Computer networks continue to improve in accessibility, speed, and coverage, leading users to rely heavily on connectivity for normal activities. However, the widely varying characteristics of networks often cause problems for their use, since applications typically assume some minimal quality of service from the network. If the network in its current state cannot provide that quality, many applications work poorly or not at all.

In many cases, more intelligent handling of data in the network could ameliorate these problems and allow applications to work well even under difficult network conditions. Active networks offer this promise by allowing substantial programmability of the network. However, most existing active network systems work on the assumption that new applications are written so that they explicitly instruct the network on how to handle their data streams. This approach offers no benefits to applications that were written before active networks were created, nor to later applications that were not written with the possibilities offered by active networks in mind. Even applications that were written for active networks are limited by the creativity and foresight of the application designer, who must become not only an expert in his own application area, but in active networking as well, to make effective use of the new possibilities. In many cases, certain sets of operations (such as cryptographic and authentication operations, lossless compression, or alternative routing) may be commonly useful for different applications. Panda could provide application writers the benefits of these operation sets when their applications work in active environments without requiring the application writers to code them for active networks.

Panda is a middleware system that provides the benefits of active networks to unaware applications. Panda traps data streams from those applications, converts them to active network packet streams, determines the network conditions, makes a plan of which adaptations to apply to the streams to deal with prevailing conditions, and deploys the code necessary to ensure proper handling of the streams. Panda is transparent to the applications it services, though of course any permanent alterations it makes in the data stream will be visible at the destination.

Consider the following scenario. Two users on portable devices are talking through an existing video phone program. One user is in his home, connected by a moderately high-speed wireless network to a base station in his house. The other is in a public place, using a telephone dialup line to connect to his office machine. Between the base station and the office machine, the communication goes over the Internet. Since the application in use may have been written with the assumption of wired networks with fairly high

[1] This work was supported under DARPA contract number N66001-98-C-8512.

and uniform speeds and bandwidths, very likely the limited bandwidth of the dialup line and the possible interference on the wireless link will cause problems for the video and audio. Further, the users may be concerned about the possible loss of privacy because their transmissions are crossing a wireless link and the untrustworthy Internet.

The audio packets could be given sufficient priority to ensure their timely delivery, the video packets could be selectively dropped to ensure that the most useful frames make use of the limited bandwidth, and all communications could be encrypted to provide privacy. However, the designers of the application did none of these things. Further, in some situations the remedies to be applied may be best applied somewhere other than at the application end points. For example, the home user's portable machine may lack the power to perform strong cryptography, while his house's base station is quite capable of doing so. Active networks could easily handle all of these problems, but this particular application was also not written with active networks in mind.

Panda provides a solution for the problems of matching legacy applications to the new power of active networks. In the prior example, Panda would automatically trap the data streams representing the video and audio. After examining the conditions of the networks and machines involved, Panda could choose adaptations to prioritize the audio, selectively drop video frames, and suitably encrypt at the proper place in the network. Doing so essentially requires that Panda automatically create a plan for determining which adapters to deploy in which locations. Panda would deploy those adaptations, convert the application's data packets into active network messages, and ensure that these messages were delivered to the Panda active network components at all participating nodes.

The model foreseen for Panda use is that a wide variety of adapters would be available for Panda's use. Some would be highly general, some quite specific to certain types of data streams or even certain applications. A general planning facility would choose the proper set of adapters to meet the prevailing conditions. If necessary, application writers or users could write new adapters to handle previously unforeseen conditions or special needs of their data streams, but even without such specialized code Panda should be able to offer useful services to many applications. When appropriate, users and application writers should also be able to offer Panda advice on how to handle their data streams. In essence, Panda would offer a useful service to users who know nothing about Panda or active networks, while allowing for even greater utility for those who do understand those technologies.

Panda is intended to run on fairly powerful nodes, since it does significant processing on packets. Panda would not be suitable for use on a core router, for example, but would be suitable for a router providing access between a subnetwork and the backbone, or on a gateway to a wireless network, or perhaps on a server machine attached to a router, assuming that relatively few of the packets passing through that router would need to be diverted to the Panda server. Panda provides significant benefits to data streams, but it does so at a cost, and thus its deployment points should be carefully considered.

This paper describes the basic architecture and current implementation of the Panda system. The paper also describes demonstrations of the efficacy of Panda and presents performance data on the system. It discusses the lessons learned during the Panda project about transparent adaptation of data streams, composition of multiple adapters, and automated planning for active networks.

2. Panda Architecture

To ease implementation, Panda is built on top of ANTS, an existing active network execution environment (EE). This EE provides Panda with basic active networking services, such as executing code at a node on behalf of a packet, deploying adaptation code to the required nodes in the network, etc. The ANTS execution environment [1] is a Java toolkit that provides a protocol-based programming model for customizing packet forwarding through a network using a data format called capsules. Simple use of ANTS typically carries the programs to be executed in the capsule along with the data and control fields. While ANTS did not perfectly match the Panda model of active networks, it required only minor alterations to support Panda.

Panda currently supports UDP-based application data streams. The underlying ANTS system makes no guarantees regarding the delivery of capsules or the order in which capsules will be received at the destination, much like UDP. Also, multimedia applications, which tend to use UDP, are good candidates to benefit from a distributed adaptation system since they put heavy demands on the network and often perform poorly under degraded network conditions, since random loss of significant numbers of multimedia packets tends to seriously degrade the quality of the video and audio. The Panda approach could be applied to TCP streams, but would require the addition of a reliable data delivery model suitable for TCP applications. [2] demonstrates that a TCP-friendly reliability model can be built at reasonable cost, but Panda does not currently include such a model.

Currently, Panda supports unicast applications only, although it has been used for simple multicast-like operations like forwarding incoming data to two different outgoing branches.

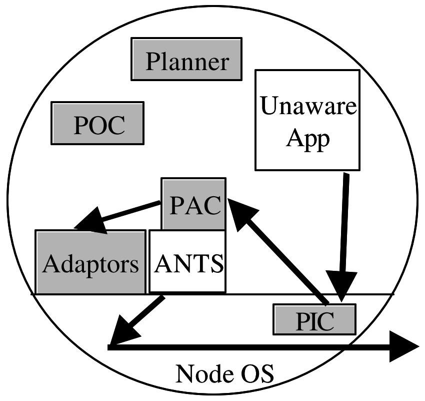

Figure 1. The Panda Architecture

The Panda architecture has four modules, each of which addresses a major task in the middleware system (Figure 1). The Panda Interception Component, or PIC, is responsible for obtaining data from clients. The Panda Adaptation Component, or PAC, deploys and runs adapters for multiple client applications. The Planner chooses a set of adapters that solve the network limitations to meet the users requirements and preferences. The Panda Observation Component, or POC, deals with gathering and reporting information required for all other Panda components, including planning. Figure 1 shows a Panda installation on a source node, with the arrows indicating the flow of a packet from the unaware application, through the PIC to the PAC, which passes it to relevant adaptors. When the adaptors are finished, they give the packet to ANTS, which invokes node operating system services to forward it to its destination. The shaded boxes are the four Panda components plus Panda adapters, the part of the system reasonably definable as Panda.

The Panda Interception Component (PIC) must intercept all data streams that Panda may wish to handle. Depending on the facilities provided by the host operating system, this interception can be accomplished in different ways. The current implementation uses a Linux loadable kernel module (LKM) to intercept socket calls. Alternatively, the firewalling capabilities built in the Linux OS could also allow the necessary redirection and masquerading of connections, or Linux IPtables could handle this problem. Systems like the x-kernel [3] and Scout [4] have built-in capabilities to control handling of network connections. Regardless of the interception mechanism used, the PIC must also have some way to know which data streams to intercept.

The Panda Adaptation Component (PAC) is the core of the Panda system. It installs the necessary adapters for a data stream, delivers capsules to the proper adapters, and generally controls the flow of a data stream through Panda nodes. Because these responsibilities heavily overlap the typical behavior of an execution environment, this portion of Panda is tightly coupled to the underlying EE.

Panda adapters are modules that accept a data packet and can perform arbitrary modifications on that packet, including dropping it or converting it into more than one packet. Panda may deploy more than one adapter for a single data stream on a particular node, so the system must allow for the output of one adapter to serve as the input for the next. Since a packet can be dropped, Panda must also allow for situations where not all adapters deployed on a node are actually invoked to handle a particular packet.

During execution, an adapter may store data at several different locations in the Panda environment. The ANTS node cache and the POC provide interfaces to store many distinct data items. The Panda system also provides an additional interface to dynamically store data within the capsule, known as the capsule cache. The content of the capsule cache is maintained as the capsule traverses the network and is available to any adapter that runs on this capsule. The capsule cache allows adapters on different nodes to add information specific to their operations to the capsule in a general and commonly known way.

The Panda Observation Component, or POC can be viewed as the central service for messaging between all Panda components (analogous to a CORBA ORB). A typical Panda node has a POC running locally. Two types of components connect to the POC: sensors and clients. Sensors generate information. Clients obtain the data generated by the sensor via the POC. In some cases a component may be both a client and a sensor to the POC; for example, a component that provides hysteresis-type functions on data to a client could obtain the original data from another POC sensor.

The Planner is the most important client of the POC in the Panda system. The Planner uses the POC to determine the current network conditions and other information needed to determine a suitable plan for an application's data stream. The Planner also can optionally accept user preferences to better tailor the plan to suit a particular user's needs. User preferences can be implemented as a POC sensor that interacts with the user, and this configuration reduces the complexity of the Planner as it only needs to be a POC client to obtain this additional information regarding the user. The Planner is a sophisticated facility that combines distributed data gathering, temporary planning at each virtual link between two Panda nodes on a data path, and a centralized planning facility that uses the data gathered from all other nodes.

Panda is capable of supporting multiple different planners. Initially, Panda used a very simple template-based planner. This simple planner has been replaced by a far more powerful planner based on heuristic search [5]. In brief, this planner uses information about the data stream, network and node conditions, and adaptor availability to search the space of all possible plans for the best plan. Heuristics based on constraints of adaptations and observations of how adaptations should be deployed allow the planner to create high quality plans in much less time than an exhaustive search would require. Despite examining less of the solution space, the Panda planner typically chooses exactly the same plan chosen by a full exhaustive search, as demonstrated by thousands of experiments under a wide variety of conditions. Planning runs on the node that initiates the data stream.

Panda, under normal conditions, works transparently using automated planning; the application programmer or user need not know anything about it. Sometimes being less transparent may be valuable. An application may be aware of the active network; it may have better knowledge of critical network and system conditions. Therefore, an application programmer can control sockets intercepted by Panda through a standard socket API. The API allows the applications to control the planning process. For example, the application may provide its own plan, or it could alter Panda's plan. Panda provides finer mechanisms to influence planning, as well.

Panda also provides a user interface so that users can set preferences for how Panda will handle their data streams. Users have the option of selecting which streams and data types to adapt and with what priority. Voice transmission may have higher priority than bulk data transfer, for example. Users can choose data fidelity levels, such as minimum tolerated image resolution. Other options include security level desired and communication delay constraints. All these preferences are used as input by Panda when it performs its automated planning.

There are other interface features that are not directly related to Panda. The application can use an API to communicate with the system to obtain the latest information about the system and network conditions. When appropriate, the application can use such information to trigger Panda replanning.

The Panda project concentrated on the feasibility of the core idea and several issues key to the notion of application-unaware use of active networks. Thus, the existing system does not address all relevant issues for an active network middleware component. First, Panda uses only the ANTS mechanism for code transport, which is not ideal for its purposes. Second, Panda does not address any security issues involved in providing a distributed adaptation service, though associated research [6] has addressed some important security issues. Third, since Panda works with UDP streams, it does not provide reliable data delivery or recovery of failed adaptors or other Panda components, though again associated research [2] addresses these issues. Finally, Panda does nothing with routing, though alternate routing policies could be beneficial.

3. Panda Implementation

3.1. Basic Implementation Details

The current Panda system has implementations of the PIC, PAC, and planning components, in addition to various adapters. The POC is under development. Panda is written in Java, with the exception of the PIC, which contains a Linux LKM (written in C) and a JNI interface to control its operation. The PIC and PAC contain approximately nine thousand lines of code, not including code for adapters. The planner consists of around five thousand lines of code, plus some code to interface the other Panda components to the planner.

Panda is built on top of a modified version of the ANTS 1.2 distribution. The most significant change to ANTS was to support larger capsules – larger in both size of code and size of the data sent over the network. Additionally, Panda required changes to the ANTS dynamic code-loading system to allow capsule code to be loaded from any node. Also, instead of being a permanent part of a particular protocol, under Panda a given adaptor may be used in many different situations, which requires alteration to ANTS dynamic code-loading, as well. These changes break the fundamental principles of how the ANTS system works, but these changes are necessary to run Panda.

Panda runs on the Linux operating system with kernels from the 2.0 or 2.2 series. It requires a JVM version 1.1 or higher. It has also run on Janos [7], using a customized version of the Kaffe VM [8]. The kernel module of the PIC needed to be reimplemented to work in the Janos environment, but the Java interface to the PIC remained the same, only requiring minor Java code changes to cope with two different interception implementations.

3.2. PIC Implementation

The current Panda PIC is a LKM stacked on top of the native networking functions to provide additional control over the proxy and masquerading facilities built into Linux. Using a kernel module for interception allows Panda to intercept any application's data stream running on the node, regardless of how the application is linked or what libraries it uses. Panda receives an application's data at the system-call level before any network-level transformations have occurred, such as segmentation or the addition of checksums. Unfortunately, this approach is subject to any user-level buffering that may occur when using standard I/O libraries. Panda also has no access to any information that is present in a user-level networking interface, if one is used.

In the case of UDP communications, the middleware opens a new UDP socket for interception and performs a LKM sockopt() informing the LKM that this socket wishes to intercept certain UDP packets. The LKM diverts any outgoing datagram that matches the intercept description from the original destination to the interception UDP socket opened by the middleware service by changing the destination address of the packet before it reaches the normal kernel networking code. The original destination address is stored in the module in a per-socket data structure. After receiving a diverted datagram on the interception socket, the middleware service issues an LKM sockopt() to obtain the packet's original destination address. At this point, the middleware is now able to send the payload over the active network.

The Panda middleware at the destination node strips the active network components from the datagram and sends the non-active datagram to the real destination application, using the LKM to masquerade as the original source. As in packet interception, the middleware makes use of a LKM sockopt() to control the masquerade address for the packet. The middleware sends the packet over a socket, and the LKM in turn makes use of facilities in the standard Linux kernel networking code to perform masquerading on the packet.

UDP communication is connectionless, so it is unnecessary for an application to send a close signal over the network to another computer. But without a close signal, the Panda system cannot reliably determine when to free resources associated with a data flow. To solve this problem, the LKM watches for UDP socket closes and sends a close signal to any interception socket that has intercepted data from the closing socket.

Interception is initially performed on UDP packets destined for well-known port numbers. Since most applications make use of well-known port numbers to reach standard services on a server, this has not proved to be a limitation. While this approach is certainly less flexible than interception based on signatures that may be found in the data stream itself, it incurs less overhead and latency to the applications that cannot receive benefit from the middleware service.

Interception can also occur on other packets or connections that are related to the application, but not on a well known port number. For instance, in a TFTP file transfer, only the initial file request is sent to a well-known port number; the data transfer and acknowledgement packets are sent to dynamically assigned port numbers chosen by the operating system. In these cases, the new port number to intercept can be determined from the source address or from information in the payload.

3.3. PAC Implementation

. The PAC is implemented as an ANTS application that handles data from multiple user applications and converts the data into capsules that are sent over the active network. At the destination, the PAC removes the data from the capsule and delivers it to the receiving application. The design of ANTS does not require a Panda data stream to pass through the PAC at intermediate nodes, even if adaptations are performed there, other than during the planning phase at the start of connection setup.

3.4. Panda Adapter Implementation

Adapters in the Panda system are placed in a special portion of an ANTS capsule, with one adapter per capsule type. This placement provides a number of benefits and also allows reuse of much existing capsule code with a minimum of changes. One of these benefits is that the loading of capsule code to a node is handled by the ANTS system. Additionally, Panda benefits from any capsule-code security mechanisms that are built into ANTS when loading capsules at a node.

In Panda, adapters have complete control over the capsule, including routing and transformation. Panda is designed to provide as much flexibility as possible in the adapters it can use. This decision also reduces the size and complexity of the Panda code resident in the capsule by delegating routing and forwarding to an adapter.

Panda creates a plan of which adapters to deploy to allow the data capsules to reach their destination and receive the special treatment required by current network conditions. When a Panda capsule begins evaluation at a node, it does not know what adapters need to be run. The *plan access method* determines which adapters a capsule should run. To support different styles of planning, there are three plan access methods built into Panda. First, the plan could be embedded into the capsule. Second, the plan could be in the ANTS node cache. (This method is used for Panda's heuristic-based planner.) Finally, the capsule can visit the planner on the current node to determine the set of adapters to run there. A capsule may try any combination of these plan access methods, depending on how the capsule was initialized. Should all of these methods fail to provide a set of adapters to run, as in the case where a capsule is forwarded along an unexpected link, a simple shortest-path forwarding routine built into the data capsule is run.

Once a set of adapters is found at a node, control of execution is transferred to the first adapter, which has complete control over the capsule. It may choose to transform the payload or headers (including the planning information), forward the capsule, or run the next adapter. The list of adapters to run is kept in memory, and the currently executing adapter can either call the next adapter in the list or terminate execution of the capsule after it has performed its functions. Most adapters will simply call the next adapter on the list until the end of the list is reached, where capsule execution will terminate. This includes forwarding/routing adapters, which should be normally placed at the end of the list of adapters to run. Adapters typically trust each other. Issues of handling adapters that do not trust each other are handled by excluding untrustworthy adapters in the planning phase.

3.5. POC Implementation

The POC must accept sensor information from various sensors, including ones that do not reside on the local node. To allow for different types of POC sensors to be built, the POC employs a common modular interface to add and query sensors. This modular interface maps neatly into the Java system. This system can also integrate with existing monitoring systems, as the POC sensor module can simply act as a bridge between the POC and the component that performs the actual monitoring.

Clients to the POC are typically other Panda components. POC clients can determine the available sensors, add and remove sensors, and obtain information from a sensor attached to the POC. Adapters can act as either sensors or clients of the POC, although because adapters are implemented as capsules, they cannot communicate with the POC without special provisions. For operations where the data is not time-sensitive, the client can get POC information and store information as a POC sensor in the ANTS node cache. Periodically, the PAC will examine the contents of the node cache and act as a proxy to the POC for the adapters. This method of communication with the POC lessens the amount of time the adapter spends performing its role as a sensor or client. The adapter also has the ability to communicate with the POC through the use of an ANTS extension. After finding the POC extension on a node, an adapter acts as any other client or sensor to the POC.

POC clients usually run on the same node as the POC. However, many clients, such as the Planner, need access to information that resides on other nodes. Thus, the POC implements a gateway module to query information that resides on a remote POC. With the module, a client asks its local POC for information residing on a remote POC, and the gateway module obtains the information from the remote POC and sends it transparently to the client on the local machine, using underlying Panda out-of-band communications facilities. The gateway module can be implemented as a standard client and server to the local POC that runs on all nodes.

The POC currently uses very simple sensors, at the moment. More sophisticated sensors could be added, at the cost of their development. A better solution would be a close integration of the POC with an existing active network sensing and management facility. In the past, Panda has been successfully attached to Nestor [9], and investigating further use of Nestor with Panda would be valuable.

3.6. Panda Planner Implementation

The Panda planner runs a simple protocol to gather all information necessary to build its plan. This protocol requires essentially one round trip from source to destination and back before all information is available to the planner, with slight extra overhead because some processing is required at participant nodes during the round trip. Thus, gathering the data and performing the heuristic search can take some time. Therefore, Panda also creates a temporary plan quickly, to allow data to start flowing before the normal planning procedure completes. This temporary plan is built on a per-node basis, with each node using purely local information from itself and the next Panda node to determine which adapters to deploy on those nodes. These temporary plans can be very far from optimal, but they allow some data to flow while the full planning procedure occurs. Because network conditions can change substantially during the lifetime of a data stream, the original plan may become ineffective, so Panda supports replanning. The mechanics of installing the new plan are essentially the same as those of switching from the temporary plan to the full plan at the start of the data stream.

3.7. Sample Panda Applications

An early application of Panda assisted in transmitting a video from a server to two destinations with differing link throughputs. Without Panda, the server would have to send a customized version of the video stream to each client to provide them with the maximum video fidelity attainable over their respective connections. With Panda, we used two adapters to achieve a better effect. The first adapter duplicated a single, original quality unicast video stream from the server and forwarded it over high-quality links to two intermediate nodes. The second adapter was run at these intermediate nodes and filtered the video stream to meet the individual throughput restrictions to the clients, who thus each received the best possible quality of service for their connectivity while reducing the throughput and computation load on the server.

A more complex application of Panda involved multiple components from UC Berkeley, the University of Utah, ISI, and Columbia University. In this scenario, a Berkeley Ninja server [10] sent a video stream accompanying a presentation to a client connected through an overloaded link. The video stream contained multiple versions of the video, each encoded at a different quality. Panda intercepted the video stream and performed two actions. First, it set up a virtual active network (VAN) from the source to the destination node using software designed by Columbia [11]. The VAN used an active form of RSVP [12] built by ISI to guarantee the throughput over the congested links. At an intermediate node running Panda and Janos [7], an adapter only forwarded the highest quality version of the video stream that the client could receive.

Another demonstration of Panda also involved interoperation with UC Berkeley's Ninja, Columbia's Virtual Active Networks and Nestor, and the University of Utah's Janos system. The scenario for the demonstration was a videoconference, with two different video/audio sessions being streamed to a third participant, in an extended Y-configuration, through a heterogeneous network with a variety of problems. Network problems included a packet storm on the wired segment, as well as extensive wireless competition. In order to deliver acceptable video and audio, network conditions had to be analyzed by Nestor [9], and the media appropriately adapted. Adaptation in this case was a selective layer-based distillation of the video, encoded in the WaveVideo wavelet codec [13], based on prioritization of the streams. Prioritization was determined by a bandwidth analysis of the audio traffic, hypothesizing that more audio traffic would indicate a speaker. Due to the tremendous number of packets from the videoconferencing sessions, the final wireless link was incapable of delivering acceptable video for both senders. Thus, Panda was required to selectively drop packets from the less desirable session, while maximizing the quality of the "focused" session. The end results were usable video streams with the higher resolution stream dynamically switched to the camera showing the current speaker.

4. Panda Performance

4.1. System Overheads

Panda puts substantial code (itself, ANTS, and adapter code) in the path of packets it intercepts. The overheads associated with this code determine the domains for which use of Panda will be beneficial.

The figures in this section concentrate primarily on the latency induced by Panda, though some data on achievable throughput is also presented. The data covers minimum possible latency with Panda, the latency effects of including multiple participating Panda nodes, and the latencies induced by adding minimal adaptors and realistic adaptors. Error bars on all figures show the value of standard error, unless otherwise indicated.

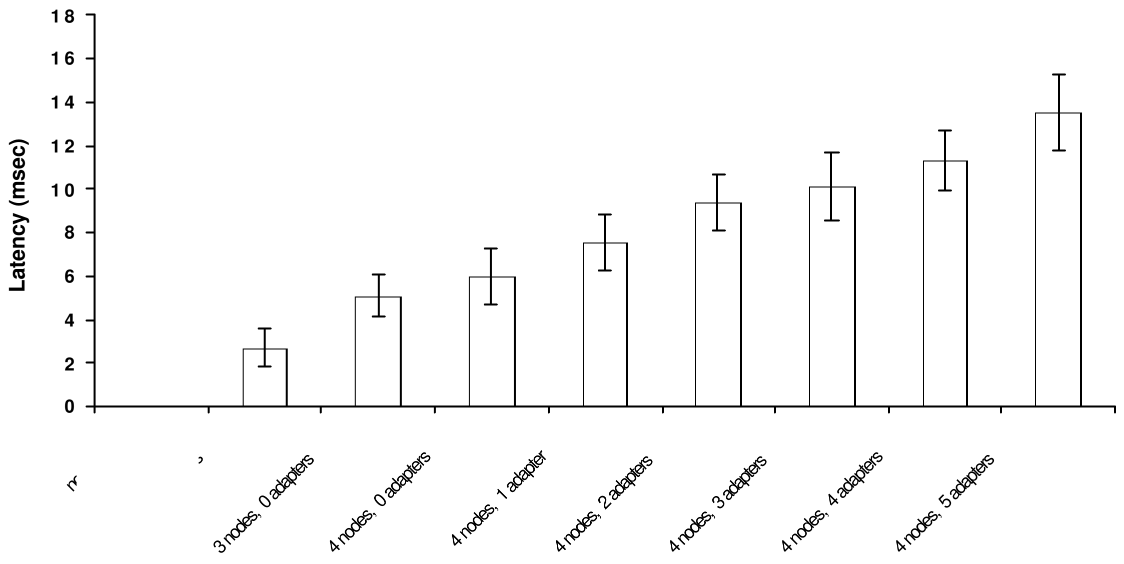

Figure 2. Packet delivery latency

One fundamental overhead is the additional latency of delivering a packet. The following method was applied to measure one-way packet latency. The packets were stamped with the local time on the source machine. Upon the arrival at the destination machine the stamped time was subtracted from the destination local time to obtain *measured time delivery*. The synchronization of the source and destination machines' clocks was done with NTP. The NTP server was located on the destination node. The source node synchronized itself to the destination local time before the first packet was sent to the destination. Then 20,000 packets were sent to the destination. After the last packet was delivered, the source machine measured the *skewing value*. It was presumed that skewing grows uniformly by time. The *actual time delivery* was calculated with a formula for each data packet *n*:

$$ActualTimeDelivery(n) = measuredTimeDelivery(n) - \frac{skewingValue}{20{,}000} ? n$$

The connection was tested with twisted-pair sequential connections of up to four computers. Dell Inspiron 3500 laptops with 333 MHz processors were used for one set of tests and Hewlett Packard Omnibook 4150 laptops with 500 MHz processors for another set of tests; all machines used Linux Red Hat 7.0 with the 2.2.16 kernel. Xircom RealPort2 Ethernet 10/100 PCMCIA cards were used for the network connection between the machines. The source and destination machines ran a user application and the Panda code concurrently. The priority of the user application was set lower on the source machine and higher on the destination machine to ensure proper allocations of resources.

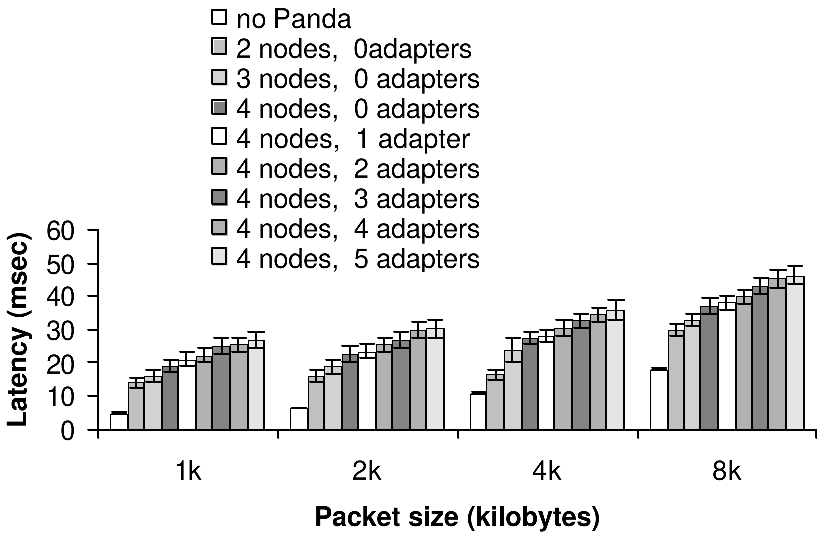

Figure 3. Null adapter latency

Throughput of the network links is varied among 150 Kbps, 800 Kbps, 2000 Kbps, and 5000 Kbps using CBQ.

Figure 2 presents packet delivery latency for different packet sizes. The packet delivery latency also contains the adaptation latency. Figure 2 shows that adding Panda to a data stream increases its latency 50 to 150%, with longer packets seeing less effect. Adding more Panda-enabled nodes or more adapters modestly increases the delay for each addition.

Figure 3 presents the latency of inserting adapters that do nothing. All adapters were deployed on a single node of the connection for each bar. Without Panda no adapters can be deployed, so the extra latency for that case is defined to be zero. Every Panda node always runs at least one *forward* adapter, whose only task is to forward a packet to the next node after all other adapters are executed. A number of forward adapters equal to the number of connection nodes is always present in a Panda connection but this is not considered in the adapter counts used on these graphs.

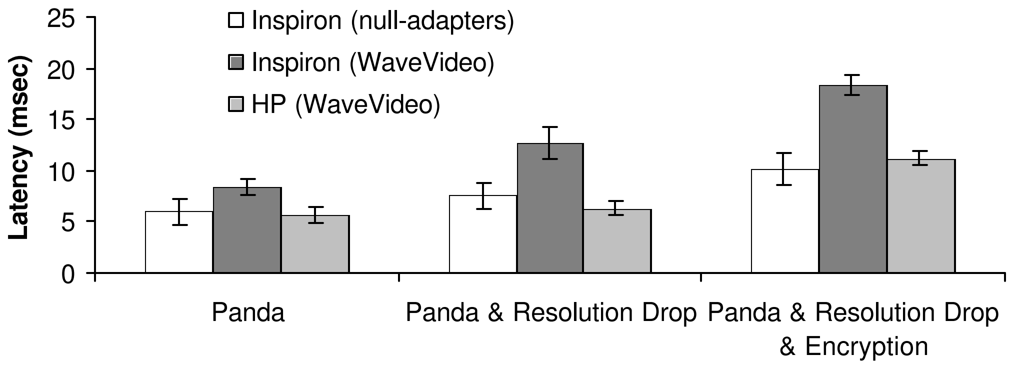

Figure 4. Latency of running real adapters

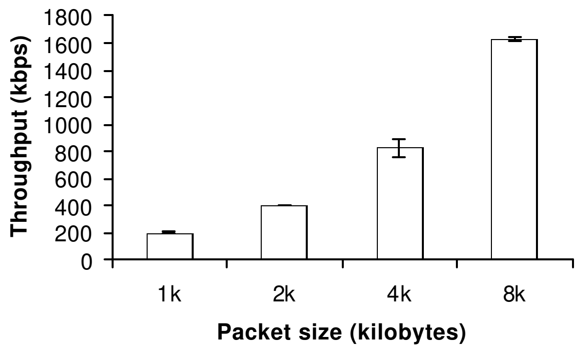

Figure 6. Panda throughput

Figure 4 presents the latency of the adaptation with real adapters. This figure and figure 5 were obtained by running a WaveVideo application on the same configuration used throughout this section, using adapters that filtered the video and/or performed encryption and decryption. Since real adaptors are often CPU-bound, more powerful machines achieved lower latency, as shown in figure 4.

Figure 6 shows how Panda throughput grows with packet size. As expected, larger packets achieve higher throughput. Error bars represent 95% confidence intervals.

The planning procedure consists of planning data-gathering, plan calculation, and plan deployment. Planning data-gathering takes one round trip; the source node forwards the data gathering message to the destination node and waits for its return. Planning data-gathering for the four Panda nodes in the test configuration takes 108 +/- 2.85 milliseconds.

Figure 6 shows the latency of plan calculation for a connection that may require no adapters, or just a Resolution Drop adapter, or both Resolution Drop and Encryptor/Decryptor adapters. The bandwidth of the links was varied, but the graph shows that plan calculation latency does not depend on the available bandwidth.

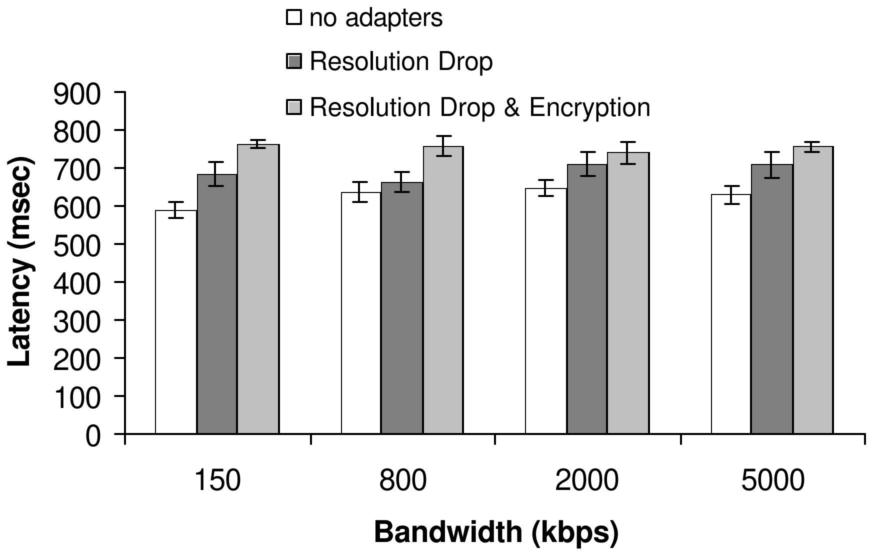

Figure 5. Plan calculation overhead

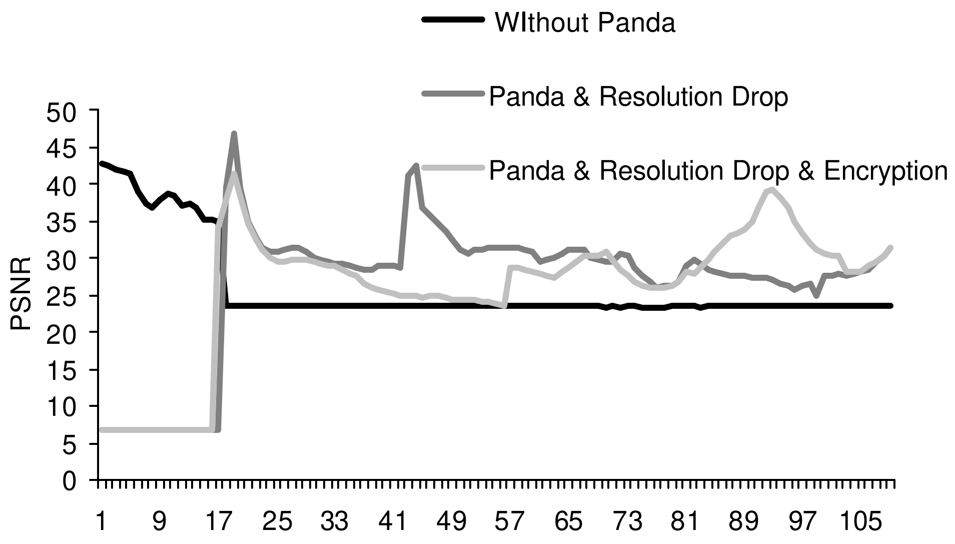

Figure 7. PSNR for Wavevideo application

The latency for deploying the adapters selected by the planner depends on adapter size and the available link bandwidth, as shown in figure 7. Resolution Drop is a very small adapter that contains a few lines of code. Encryption is a heavyweight adapter that processes every character of user data to perform DES encryption. The larger the adapter, the longer it takes to deploy it. The deployment latency does not depend on bandwidth unless it is less than 150 Kbps.

4.2. Panda Benefits

Panda is worth using only if the benefits it offers outweigh the overheads. For some benefits, such as encryption, quantifying the benefit is hard, particularly for purposes of comparison to latency overheads. Here we

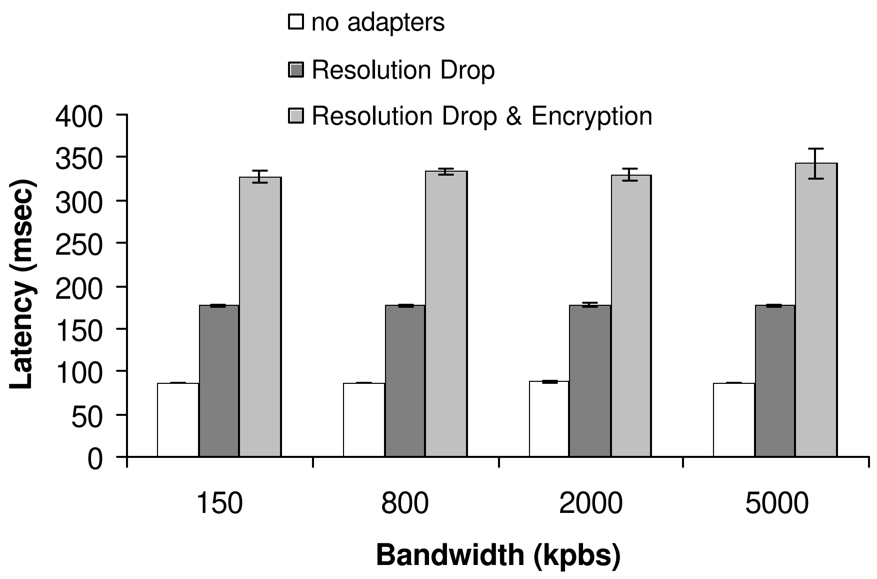

Figure 8. Plan deployment latency

present benefit metrics that are more quantifiable and take the latency overheads into account. In particular, we present improvements in the Peak Signal-to-Noise Ratio (PSNR) for the WaveVideo application discussed earlier. Figure 8 presents PSNR luminance on Dell Inspiron 3500 machines with a link bandwidth limited to 150 Kbps.

Without Panda, the PSNR curve declines when the channel is saturated and more or less random video packets are dropped. Panda, using the Resolution Drop adapter, intelligently adjusts to the limited bandwidth by dropping packets representing lower resolution video components. As a result, once Panda has completed its planning phase and deployed its adapters, its PSNR curve improves and exceeds the non-Panda curve. The PSNR performance of Panda with Resolution Drop and Encryption adaptation in some areas can be even better than Panda with Resolution Drop only; this is due to helpful buffering effects caused by the extra delay of encryption.

5. Related Work

Panda is the intellectual descendant of Conductor [14]. Conductor is a TCP-based open architecture framework providing a distributed, coordinated adaptation facility. Similar to Panda, Conductor supports application transparent interception and distributed, coordinated adaptation of the network stream. Unlike Panda, Conductor offers an extensive security model, as well as a reliability model designed for adaptation called *semantic segmentation*. As Conductor is a TCP-based framework, the adaptation library for Conductor is substantially different than Panda's, focusing on HTTP, POP, and other stream-based adaptations.

The Protocol Boosters [15] adaptation framework provides a general approach to network-level adaptation. The framework allows either a single adaptation module or a pair of modules to be transparently deployed, adding new features to existing protocols, such as forward error correction or fast retransmission. Boosters typically provide lossless adaptation, since the system provides no support for ensuring reliable delivery if packets intended for delivery are generated, dropped, or permanently altered by a booster. Boosters are composable, but the system does not provide support for selecting a set of boosters that will perform well together. Panda substantially differs from Protocol Boosters in its planning capabilities, as well as in its support for lossy adaptation.

Transformer Tunnels [16] use IP tunneling to alter the behavior of a protocol over a troublesome link. Once created, a transformation function is applied to all data flowing through each tunnel. Generally, Transformer Tunnels are used to provide protocol-independent adaptations, such as consolidation of packets, scheduling of transmissions to preserve battery power, encryption, lossless compression, and buffering. Transformer Tunnels are transparent to applications and may be interoperable with application-level adaptation provided by proxies. However, no mechanism is provided to compose transformation functions or to coordinate transformations with externally provided adaptations. Panda's adaptor model allows this composability; additionally the Panda Planner coordinates various adaptations across multiple links.

Proxies are often used to handle single troublesome links, particularly links close to client nodes. One of the most advanced proxy solutions is the Berkeley proxy [10]. This system uses cluster-computing technology to provide a shared proxy service for a wide variety of PDAs. The proxy can provide a variety of application-level adaptations, including transformation (changing the data from one format to another), aggregation (combining several pieces of data into one), caching, and customization (typically converting a data format for use by a particular PDA). The Berkeley researchers have investigated methods of composing adaptations on a single machine [17]. They have also examined the use of a clustered proxy service to provide highly reliable and scalable services to a large number of customers. In particular, their proxy technology has been deployed for large-scale, real-world use, supporting palm-computer based web browsing in a metropolitan-area wireless network [18]. The Berkeley Proxy and other proxy solutions typically work at a single location in the network, while Panda is designed for distributed adaptation at multiple locations.

CANS used a different approach to provide an early form of automated planning [19]. CANS performs dynamic deployment of transcoding components (similar to Panda adapters). These components use high-level specifications of component behavior and network routing characteristics as inputs, ensuring that composed adaptations are proper through the use of strong typing of the inputs and outputs of those adaptations. The CANS algorithm is based on search in a stream-type graph with a simplification strategy to reduce the graph's complexity.

6. Conclusions

The Panda project has demonstrated that active network technology can be applied usefully, even to applications that were not written with active networks in mind and that are not altered to work with active networks. This demonstration substantially increases the potential audience for the improvements offered by active networks. Not only are legacy applications potential users of active networks, but future programmers can concentrate on the needs of their applications, rather than the complexities of programming an active network. Where suitable, they can provide hints and direction to Panda or a similar system, but they can still expect that the active network will perform beneficial actions on their data streams even without such advice.

Panda achieves reasonable performance despite being unoptimized and running on an early version of ANTS, which is known to have poor performance. Even with these disadvantages, realistic applications receive measurable user- and application-visible benefits from Panda. In a more optimized form, Panda could provide greater benefits to a wider range of applications.

Panda's architecture is well suited for partial deployment of active networks. Panda must run on the source and destination node (though further development could remove even those restrictions), but otherwise does not require intermediate nodes to participate in the active network. Of course, non-participating nodes cannot perform useful adaptations, but this approach allows selective deployment of Panda at nodes that are close to troublesome links, or that often are overloaded, or that have other characteristics suggesting that they are a good spot for adaptation. The more such nodes deployed, the more options available to Panda.

Panda has also demonstrated that automated planning of active network adaptations is possible and efficient. Panda's automated facility plans sufficiently quickly to provide a plan early in most data streams, and the plans provided are usually as good as those found by exhaustively testing all possibilities. Without a reasonable planning facility, the Panda approach could not be used in the real world, so this demonstration is key to its future success. Further, this result suggests that automated planning based on a heuristic search or other AI techniques might have a wider applicability in solving many distributed systems problems.

A final lesson from the Panda project is that early choices can have long-lasting implications. The decision to build on an existing execution environment (rather than creating a new one), and the choice of ANTS for that EE, had profound implications for the project. ANTS was not designed for a model of dynamic composition of shared adapters, potentially a new set for each connection. Therefore, much of the Panda implementation effort was spent making simple concepts fit into a framework that wasn't designed to support them. The choice had other implications, such as mandating an early commitment to performing the work in Java. This choice was not a mistake, since the resulting system demonstrated all the hypotheses of the original project, but it did have wide-ranging effects on the work, many of which were not foreseen when the decision was made. For example, the combined performance overheads of ANTS and Java limited the sorts of adaptations and applications that we examine for Panda. Since running any adaptation would be expensive, only adaptations with major payoffs were worth considering and only applications with major problems were candidates.

Performance is a key weak point of Panda, as it currently stands. Even with rather high overheads to overcome, Panda is useful for many applications. However, a re-implementation that divorces it from ANTS and allows it to use a much lighter-weight EE would expand Panda's utility.

In summary, Panda demonstrates that application-unaware use of active networks is possible and can provide substantial benefits to applications. The automatic planning capability implicit in the idea can be realized with sufficiently low overhead and very high quality in the resulting plans. Thus, active network deployment need not be dependent on the creation of large numbers of active network applications. A simple piece of middleware like Panda can provide active network benefits to existing applications without altering any of their code.

References

[1] D. Wetherall, J. Guttag, and D. Tennenhouse. "ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols." *Openarch 98*, 1998.

[2] M. Yarvis, P. Reiher, and G. Popek. “A Reliability Model for Distributed Adaptation.” *OpenArch 2000*, March 2000.

[3] N. Hutchinson and L. Peterson. "The x-kernel: An Architecture for Implementing Network Protocols." IEEE Transactions on Software Engineering, vol. 17, no. 1, January 1991.

[4] D. Mosberger and L. Peterson. "Making Paths Explicit in the Scout Operating System." Proceedings of the Symposium on Operating Systems Design and Implementation, October, 1996.

[5] P. Reiher, R. Guy, M. Yarvis, and A. Rudenko. “Automated Planning for Open Architectures.” *Openarch00*, March 2000.

[6] J. Li, M. Yarvis, and P. Reiher. “Securing Distributed Adaptation.” *Computer Networks*, Special Issue on Programmable Networks, vol. 38, no. 3, 2002.

[7] P. Tullman, M. Hibler, and J. Lepreau. "Janos: A Java-Oriented OS for Active Networks." IEEE Journal on Selected Areas of Communications, Vol. 19, No. 3, March 2001.

[8] G. Back, W. Hsieh, and J. Lepreau. “Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java.” Fourth *Symposium on Operating Systems Design and Implementation (OSDI 2000)*, October 2000.

[9] Y. Yemini, A.V. Konstantinou, and D. Florissi. "NESTOR: An Architecture for Self-Management and Organization." IEEE Journal on Selected Areas of Communications, Vol. 18, No. 5, May 2000.

[10] A. Fox, S. Gribble, Y. Chawathe, E. Brewer, P. Gauthier. “Extensible Cluster-Based Scaleable Network Services.” *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP‘97)*, Saint-Malo, France, October 1997.

[11] Y. Yemini and S. da Silva. "Towards Programmable Networks." IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, 1996.

[12] B. Braden. ARP project web page, http://www.isi.edu/div7/ARP/ARP.

[13] G. Fankhauser, M. Dasen, N. Weiler, B. Plattner, B. Stiller. “WaveVideo — An Integrated Approach to Adaptive Wireless Video.” *Mobile Networks And Applications (Special Issue on Adaptive Mobile Networking and Computing)*, 4(4):255-271, December 1999.

[14] M. Yarvis, P. Reiher, G. Popek. “Conductor: A Framework for Distributed Adaptation.” *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems (HotOS VII)*, Rio Rico, Arizona, March 1999.

[15] D. Feldmeier, A. McAuley, J. Smith, D. Bakin, W. Marcus, T. Raleigh. “Protocol Boosters.” *IEEE Journal on Selected Areas in Communications (Special Issue on Protocol Architectures for 21st Century Applications)*, 16(3):437-444, April 1998.

[16] P. Sudame, B. Badrinath. “Transformer Tunnels: A Framework for Providing Route-Specific Adaptations.” *Proceedings of the USENIX Annual Technical Conference*, New Orleans, Louisiana, June 1998.

[17] S. D. Gribble, M. Welsh, E. A. Brewer, and D. Culler. "The MultiSpace: an Evolutionary Platform for Infrastructural Services." Proceedings of the 1999 USENIX Annual Technical Conference, Monterey, California, June 1999.

[18] A. Fox, I. Goldberg, S. Gribble, D. Lee, A. Polito, E. Brewer. “Experience With Top Gun Wingman: A Proxy-Based Graphical Web Browser for the USR PalmPilot.” *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, Lake District, UK, September 1998.

[19] Xiaodong Fu, Weisong Shi, and Vidjay Karancheti. “Automatic Deployment of Transcoding Components for Ubiquitous, Network-Aware Access to Internet Services.” New York University Computer Science Department Technical Report CS-TR-2001-814, March 2001.