Constructing End-to-End Paths for Playing Media Objects

Akihiro Nakao, Larry Peterson, and Andy Bavier
{nakao,llp,acb}@cs.princeton.edu
Department of Computer Science
Princeton University

Abstract

This paper describes a framework for constructing network services for accessing media objects. The framework, called *end-to-end media paths*, provides a new approach for building multimedia applications from component pieces. Based on input from the user and resource requirements from the media object, the system first discovers the sequence of nodes (end-to-end path) that both connect the source device to the sink device and possess sufficient resources to play the object. It then configures the individual nodes along this path with the modules (path segment) that implement the service.

1 Introduction

The convergence of computing and consumer electronics is resulting in a rich space for designing new services. It is easy to imagine a user wanting to play a remote MP3 file on a local stereo, display the output of a digital camcorder on a TV set in another room, or dispatch an incoming phone call to a mobile headset. While products that implement point solutions in this space are now emerging, we recognize the need for a general framework that makes it easy to construct new services by logically connecting arbitrary devices.

This paper describes such a framework—it defines an architecture for constructing *end-to-end media paths*. These paths allow users to access a wide range of media objects (MPEG video, MP3 audio, computer games, JPEG images, HTML pages) from an assortment of devices (PDAs, set-top boxes, desktop workstations, game consoles), where the devices are connected to media servers, and each other, with a variety of transmission technologies (Bluetooth, IR, FireWire, S/P-DIF). End-to-end paths are a natural extension of Scout paths [11]—which connect an input device to an output device on a single node—to a networked environment. Like Scout paths, end-to-end paths encapsulate both the code needed to move a media object from one device to another (and possibly transform it in the process), and the distributed resources needed to perform this work.

One way of viewing end-to-end paths is that they introduce several degrees of freedom to the current method of accessing media objects on the web. Typically, the URL for an object implies the "service" that is to be provided—it defines both the protocol used to retrieve the object across the network and the application to run on the sink node. For example, the URL http://www.server.org/movie.mpg is a service definition that specifies HTTP is to be used to retrieve file movie.mpg from source www.server.org, and processed by whatever application program the browser has bound to the suffix .mpg. This approach suffers from several limitations, which end-to-end paths are designed to address.

- The output device is assumed to be on the same node as the browser. In contrast, our approach allows the user to select an arbitrary output device from a control program running anywhere in the network. It also allows a compound media object (e.g., a movie with audio and video components) to be played on multiple sink devices, subject to synchronization restrictions.

- The service is limited to the access protocol running on the source and sink nodes, and the application program running on the sink node. The service does not say anything about the intermediate nodes between the source and sink. In contrast, the "service definition" employed by end-to-end paths also includes code that runs on any intermediate nodes that the media object must traverse.

- The client program is assumed to be a monolithic application. In contrast, our approach allows the code running on the source, the sink, and all intermediate nodes to be constructed from building block components. This makes it easier to adapt the media path to the specifics of a media object and the currently available resources.

This paper defines an architecture for end-to-end media paths, and in the process, illustrates how the architecture

can be used to build an example service. It also briefly describes a prototype implementation of the architecture in the Scout operating system. As this work represents only a first attempt to harness the capabilities of a pervasive computing environment, its main contribution is to define a new way of framing the problem of how to build network applications.

2 Design

This section introduces the problem we are addressing, describes our high-level design decisions, and identifies the limitations in our approach.

2.1 End-to-End Paths

Consider a computing environment that contains a collection of *media objects* and a network of interconnected *nodes*, each of which supports a set of *devices* that can be used to create, store, display, record, or transmit media objects. Given such an environment, suppose a user wants to stream an MPEG video from a server to a nearby display. Depending on the resources available at the display device and within the network, several scenarios are possible. If the network has sufficient bandwidth, and the display device has high enough resolution and enough cycles to run a software MPEG decoder (or contains decoding hardware), then we can simply stream the video to the device. However, suppose these resources are not available—for instance, the network cannot forward the entire stream due to a low-bandwidth wireless link, or the display device does not have a fast enough CPU. In such circumstances it may still be possible to watch the video with assistance from the server or the network's intermediate nodes. For instance, if there is not enough network bandwidth, an upstream node may thin out the MPEG stream by selective frame dropping; if the display device is not powerful enough to fully decode the video, the upstream node may transcode the video into a format or resolution that the display can handle.

Employing intermediate network nodes to help process the data stream is merely the next step in the evolution of application programs. Traditionally, an application was a single program running on a single machine. With the proliferation of networks, applications were often split into two components: a client program running on the user's machine (data sink) and a server running elsewhere (data source), with the network providing a bit-pipe between the two components. The next step, as illustrated by the above example, is to distribute portions of the program among the nodes that connect the source to the sink. All of these components, working in concert to implement an application, form an *end-to-end path*.

This paper focuses on end-to-end paths for delivering streaming media, and in this context, proposes an architecture based on the Scout OS. We select Scout as the OS running on each node because it provides explicit support for *modularity* and *encapsulation*. First, the code running on each node on an end-to-end path forwards the data to the next node (or to the eventual sink device) while simultaneously applying some transforming filter to the data. We view such a service as adding functionality to the forwarding path, and so provide a linear, modular representation of the path in which code modules can be replaced and new ones added. This gives the path programmer flexibility in composing a new service and the leverage of a standard code base. Second, encapsulating these modules and their associated resources within a Scout path provides a means of accounting for resources consumed by the data flow, as well as giving the system information it needs to do resource scheduling, that is, to provide QoS guarantees or avoid cross-talk between data flows.

In summary, an end-to-end path is defined by three elements: (1) a route (sequence of nodes) through the network that connects a source device to a sink device, (2) the code modules on each member node needed to both forward and transform the object, and (3) the resources (e.g., cycles, bandwidth) needed on each node to forward/transform the object.

2.2 Problem

The problem of creating an end-to-end media path across a set of nodes is one of resource discovery and management. First, we must determine what resources are available in the network and map an end-to-end media path that satisfies the user's requirements onto these resources. Second, we must program the individual nodes along the path to implement the necessary services. Our architecture reflects this hierarchy by dividing the problem space into two distinct levels:

- At the *global* level, end-to-end media paths are mapped onto the collection of nodes and devices in the network. This involves resource discovery throughout the network, and choosing a route for the end-to-end path that includes the resources it requires.
- At the *node* level, a piece of an end-to-end path is instantiated on a particular node. Here we program the node to implement one *segment* of the end-to-end path. This involves loading code modules and binding them to particular devices and packet flows.

Figure 1 illustrates an example path that spans three nodes.

In a sense, these two levels correspond to two problems that are usually treated separately: the first can be viewed as QoS routing and the second can be viewed as a form of active networking. Although our architecture exploits this separation, a major contribution of our approach is to demonstrate how these two components are interconnected; how to derive the code that should run on each node

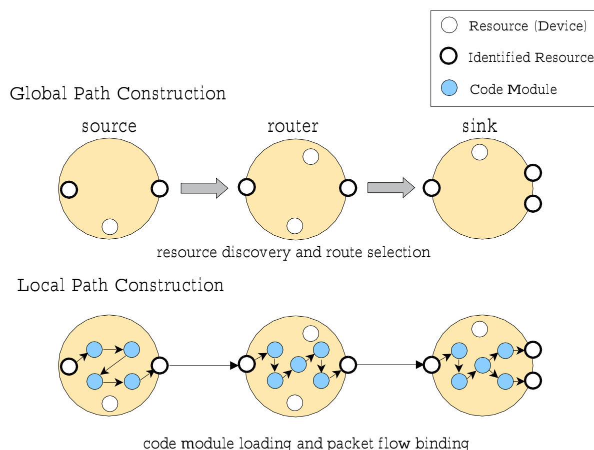

Figure 1: End-to-end path construction

from QoS considerations. A second contribution is to account for four independent sources of information required to construct an end-to-end path. The first is the media object itself, which specifies the network resources needed to access (play) the object. The second is the user, who may have some preferences about how he or she accesses certain types of media objects. The third is the set of nodes, which know their own capabilities (i.e., the devices and other resources they possess). The fourth is the "path programmer" who defines a service by giving the set of rules that govern how an end-to-end path is constructed.

Section 3 describes an architecture that takes all four sources into account to determine an appropriate path. It consists of three components: (1) a *user control point* (UCP) from which users indicate what objects they want to access and where they want them output, (2) a collection of objects that record certain facts about the environment and services that can be established, and (3) a path manager running on each node that is able to instantiate the local segment of an end-to-end path.

2.3 Limitations

We adopt three simplifications to make the problem tractable. First, we assume a single trust domain. This means we assume code is downloaded from a trusted code server, with only integrity checks to ensure that this is in fact the case. In general, it might be desirable to allow untrusted code to run within an isolated sandbox.

Second, we ignore the issue of scale, focusing instead on a localized environment. However, our architecture does permit a hierarchical solution in which a local user is able to access remote media objects by treating the gateway at the boundary of the local region as a proxy for all remote objects accessed from within the region.

Third, our prototype employs a centralized solution in which the UCP collects the necessary information from the four independent information sources, selects a feasible end-to-end path, and then instantiates segments of the path on each selected node. However, this is not a limitation of the architecture, which also permits a distributed solution in which each node decides for itself how to extend the end-to-end path, much like routing is done in the Internet. We briefly discuss the ramifications of a distributed implementation in Section 3.2.3.

3 Architecture

This section describes the key elements of our framework, and outlines the process by which end-to-end paths are constructed. The main value of this section is that it teases apart the various factors that go into path construction.

3.1 Information Sources

We begin by defining the data objects employed by our system. These objects correspond to the four sources of information that influence the end-to-end path: the media object, the user's preferences, the available nodes, and the programmers that specify network services.

Throughout the discussion, we use *attributes* to describe the properties of devices, nodes, and so on. Although there are many ways to formulate attributes, we encapsulate attributes in Java objects in our prototype implementation.

3.1.1 Media Objects

Embedded in each piece of content is an instance of the following MediaObject:

```
class MediaObject
{
    ObjId oid;
    MediaType type;
    Location loc;
    int bandwidth();
    int delay();
    int loss_rate();
                    :
}
```

where ObjId gives a unique id for the object and the MediaType is given by the set of available MIME types (e.g., video/mpg).

Additional members specify other attributes of the media object, such as the network capabilities (e.g., bandwidth, delay, and loss_rate) required to transmit the media object, as well as a specification of the output device(s) (e.g., display_res, audio_chan) needed to play the object. One of the attributes, Location, specifies the node and device where the object is stored.

```
class Location
{
    NodeId nid;
    DevId did;
}
```

3.1.2 User Preferences

We assume each user provides a profile that defines his or her preferences for devices that are used to activate (play) certain types of media objects. These preferences are specified by UserPref object as a set of attributes. While these preferences are static in our prototype implementation, it is easy to imagine the preferences being dynamically evaluated, for example, GPS or InfraRed trackers could be used to record the user's current location. [7]

3.1.3 Node Objects

A node object (Node) records information about the set of devices available on some node in the system.

```
class Node
{
    NodeId nid;
    Device[] dev;
    boolean hasDevice(VirtualDevice vdev);
    int cycles();
    int bandwidth();
              :
}
```

where NodeId is a unique id for the node, Device is the set of devices available on the node, and the other fields represent additional properties of the node (e.g., available CPU cycles, bandwidth of in-coming and out-going links, and so on). Each Device, in turn, is defined as follows:

```
class Device
{
    DevId did;
    DevType type;
    DeviceDriver dd;
}
```

where DevId is a unique id for the device and the DevType field identifies the type of the device, which is one of the set { network, storage, video, ... }. The DeviceDriver member gives the code to enable the device. Subclasses of Device may define other device-specific attributes, such as a NIC's IP address or a display's resolution.

```
class DisplayDevice extends Device
{
    Resolution resolution();
    int color_depth();
}
```

3.1.4 Rule Objects

A set of *rules* define the end-to-end paths that can be constructed. Programming a network service involves defining one or more of these rules. Although the collection of rules can be viewed as a set of macros, with the resolution process being essentially a macro expansion, it is more instructive to view the rules as existing at two levels. At the global level, *end-to-end* rules describe the behavior of a path across a sequence of nodes. At the local level, *per-node* rules describe the behavior of a single node along the end-to-end path.

At the global level, a PathRule defines the end-to-end behavior of a path:

```
class PathRule
{
    MediaType type;
    Action a;
    Template[] t;
}
```

indicating that one of the specified Templates is appropriate for applying the given Action (e.g., play, display, listen) to an object of the specified MediaType. One element of the set of Templates is eventually selected, depending on the resources available on nodes in the network.

Each Template in the PathRule specifies a sequence of node behaviors, where an individual node's behavior is given by a NodeRule. Specifically, each Template is given by a regular expression, defined according to the following syntax;

abc	(a string that names a NodeRule)
m n	(concatenation; m followed by n)
$\star$	(repetition; zero or more times)
+	(repetition; one or more times)
|	(alteration)
?	(option)
(.)	(bracket)

For example, there might be a PathRule with MediaType=video/mpg, Action=play, and the following set of Templates:

```
{  mpg_src ip_fwd* mpg_sink,
   mpg_src ip_fwd* xcode ip_fwd* mpg_sink }
```

This example says that two possible end-to-end paths connect an MPEG source device to an MPEG sink device. The first passes through zero or more IP routers (ip_fwd), while the second passes through zero or more IP routers, a video transcoder (xcode), and zero or more IP routers.

At the local level, each NodeRule is defined by the following object:

```
class NodeRule
{
    RuleName name;
    MediaType type;
    VirtualDevice[] vdev;
    boolean match(
        Node n,
        MediaObject m,
        UserPref p);
    void transform(
```

```
        MediaObject m,
        UserPref p)
    NodeTemplate[] templates(Node n);
}
class VirtualDevice
{
    DevType type;
    boolean match(Device dev);
}
```

This object indicates what devices (VirtualDevice[]) are expected to be available on this particular node in the end-to-end path. Each VirtualDevice, in turn, defines the requirements for this device, including its DevType. The match() method tells you whether the specified Device matches the requirements or not.

The match() method in NodeRule is called to determine whether or not the NodeRule identified in a regular expression matches a particular node in the network, subject to the constraints of the MediaObject and the UserPref. In other words, this routine determines if a particular node has sufficient resources to satisfy the requirements of the media object and the user preferences.

The transform() method in a NodeRule object modifies the MediaObject to reflect the transformations that would take place on the media object if it were to pass through a node with the associated behavior. For example, if a NodeRule indicates that a video-thinning module is to run, then the transform method might modify the bandwidth attribute of the MediaObject since after a media object passes through a node running such a transformation, its bandwidth requirement has been reduced.

Finally, the NodeTemplate field of the NodeRule specifies the code that is to run on the node, should it be selected to be part of the end-to-end path. The code is identified as a sequence of modules that comprise a forwarding path. Each element of the NodeTemplate either directly identifies a code module that is to run on the node, or it indirectly names a module by specifying a virtual device. Allowing virtual devices facilitates a late binding of node-specific devices, and hence, allows the path programmer to construct node templates for generic nodes, with the details required for a specific node to be filled in once a particular node has been selected.

Continuing our running MPEG example, the NodeRule required for the sink node of the MPEG end-to-end path might be as follows:

```
class mpg_sink extends NodeRule
{
    // virtual_devices: net, audio_out, video_out
    boolean match(
        Node n,
        MediaObject m,
        UserPref p)
    {
            return(n.hasDevice(net)
                && n.hasDevice(audio_out)
                && n.hasDevice(video_out)
                && (n.cycles() ≥ cycles(m)));
    }
    void transform(MediaObject m, UserPref p)
                {// do nothing }
    NodeTemplate[] templates(Node n)
    {
            // (1) net/ip/udp/rtp/mpeg/video_out
            // (2) net/ip/udp/rtp/mp3/audio_out
    }
}
```

The sink node requires a network device as well as devices for outputting video and audio. The CPU cycle requirement is calculated by the MediaObject's helper method cycles(). This kind of boolean subexpression commonly represents resource requirements. The last two comment lines in the templates() method are examples of the sequences of code modules (local path segments) that would be instantiated on the node to play the video. Note that the logical name at the ends of the module list always matches the virtual device, which in this case is one of net, video_out, and audio_out.

3.1.5 Object Repositories

As already indicated, a MediaObject is embedded in every media object in the network, and a UserPref object is co-located with the UCP running on behalf of every user in the system. In addition, there is a *path database* that serves as a repository for all the PathRule and NodeRule objects in the system. To create a new service, a "path programmer" installs the appropriate rules in the path database. When given a MediaType and Action as input, this database returns the PathRule bound to this pair, along with all the NodeRules named in the Templates associated with the PathRule.

Each node in the network monitors its own state and maintains an up-to-date Node object. Whether or not there is a repository for these objects depends on whether the centralized or distributed resolution mechanism is employed. Our prototype implementation uses a centralized approach, which assumes each node periodically forwards its Node object to a central *node database*. This database then answers three kinds of queries:

1. When given a particular node's NodeId, the node database returns the set of Device objects connected to that node.

2. When given a VirtualDevice specification (DevType and a qualifying set of required attributes), the node database returns the NodeId for every node that contains a device that satisfies the requirements. (This is done with assistance of the helper method hasDevice() in Node object.)

3. When given a pair of NodeIds, the database returns the set of k shortest routes (ordered list of the intermediate nodes) that connect the pair of nodes. The node database is able to determine the viable routes using the network number portions of the IP address(es) recorded for the network devices of each node.

If we were to use a decentralized path resolution mechanism, routing would occur incrementally according to some external routing mechanism, with each Node object known only on that node. We return to the issue of distributing the resolution process in Section 3.2.3.

3.2 Path Resolution

Our centralized scheme for instantiating a viable path for object delivery involves two stages:

1. Global path construction (node selection)
   - locate source node (usually this process is the same as the location of the object)
   - locate a sink node that has the device(s) needed to perform the specified user action on the object.
   - locate a route (sequence of intermediate nodes) that have enough resources to enable delivery of the object.
2. Local path construction (determine path segment for each individual node)
   - resolve device references
   - download and initialize code modules on each node in the path.

3.2.1 Global Path Construction

We assume users identify media objects they wish to access through some mechanism outside our architecture (e.g., browsing the web), and that as part of this process, they specify the action they want to perform upon the object. The first step is to determine the sequence of nodes along the end-to-end path.

Selecting the Source Node: The UCP first retrieves the MediaObject associated with the selected content. It then queries the path database to learn the PathRules that matches the specified MediaType and Action. At this point, we restrict our search to those Templates in the PathRule for which the very first NodeRule in the regular expression matches the source node. The match() method of NodeRule for the source node typically looks like the following:

```
boolean match(Node n, MediaObject m, ...)
{
    return (m.loc.nid.equals(node.nid)
        && ... );
}
```

This routine extracts the NodeId (in Location) from the MediaObject to look for the source node where the media object is stored. It also knows the set of Devices available on the source node, in particular, the source device where the object is stored is specified in the DevId field of the MediaObject.

Selecting the Sink Node: The UCP expands each feasible Template in the PathRule to learn the NodeRule required of each node on the path, with the objective of learning the NodeRule for the sink node. In particular, the UCP needs to learn what VirtualDevices are available to serve as the ultimate target device(s) required to play the media object. The UCP now contacts the node database, submitting the NodeRule for the sink node. From this information, the node database is able to return a candidate set of sink nodes. This list is pruned according to the UserPref object, and then presented to the user, who then selects the ultimate sink node (devices) manually. At this point, the UCP has a set of feasible Templates and concrete source and sink nodes.

Routing: The next step is to determine the possible routes between the selected source and sink nodes. Our current prototype performs a k-shortest path algorithm [9] using the number of hops between the source and sink nodes as the metric. The UCP also retrieves the Node object for each node along the k routes.

Pattern Matching: Finally, the UCP attempts to match the set of Templates—a regular expression of NodeRules—against the nodes that lie on the k shortest routes. The algorithm first constructs a deterministic finite automata (DFA), where each NodeRule in the Template corresponds to abstract symbols in the DFA's alphabet. Whether or not a given node matches a symbol is determined by the corresponding NodeRule's match() method.

Note that although this machinery is deterministic in terms of the NodeRules, the pattern matching algorithm is non-deterministic since multiple NodeRules can match a single Node. The machinery must remember the sequence of transitions that it uses to reach an accepting state, because that determines the assignment of rules to nodes that will be used to configure the end-to-end media path. Also, since it is possible that multiple templates match, the one resulting in the best MediaObject (after transformations) is selected.

3.2.2 Local Path Construction

At this stage, the UCP knows all the nodes along the end-to-end path that satisfy the requirements of the PathRule. The next step is to configure the individual nodes along

this path, using the NodeTemplates associated with the NodeRule that matched each node.

Resolution of Virtual Devices: Before we can instantiate the template on a given node, we must bind the virtual devices in the template to actual devices on the node. The corresponding Node object contains the DevType and other attributes for each device on the node. We substitute the device driver name from this object into the template in place of the generic device (e.g., net_in/ip/net_out becomes tulip/eth/ip/eth/tulip) to create a fully-specified path. We also annotate the path with the DevId for the input and output devices to which the ends of the path should be bound.

Initialize the Path: All that remains is for the UCP to request that the path manager running on each node instantiate the specified path. This request includes the path's resource requirements—contained in the matching NodeRule for the node, and originally derived from the MediaObject—and a local admission control mechanism decides whether to allow the operation to complete. To complete the operation, a node may have to contact a code server to download one or more of the modules on the path. The path manager reports back to the UCP whether or not it was successful. Once the UCP receives a success notification from all the nodes on the end-to-end path, it signals to both ends of the path to indicate that the media object can now be played.

3.2.3 Distributed Resolution

Although centralized path resolution mechanism might be practical for a localized computing environment, it suffers from two problems. First, it is possible that the UCP will make its decision based on stale information. In contrast, deciding how to extend a path on a node-by-node basis ensures the most current local information is available at each decision point. Second, the design does not allow for the path to adapt to changing conditions over time; once a path is selected, the route is effectively locked in. In the worst-case, the system does not facilitate transparent recovery from failures.

Distributing the path selection process across the nodes in the network addresses the first problem, and is half of the solution to the second problem (the other half is permitting the path to change once a better route is discovered). Fortunately, the architecture allows for path resolution to be distributed; we discuss this case below. Unfortunately, it is not a simple matter to change a path once it has been established; we leave this issue to future research.

Distributing the path selection process requires each node to maintain an up-to-date instance of its node object. Global path construction works as follows. The process of selecting the source node works as before, except the UCP contacts the source node directly to confirm that it supports the named media object. Selecting the sink node means the UCP must either send a limited broadcast to nearby nodes looking for suitable output devices, or there must be a cache of local nodes to consult. The main change comes in the Routing and Pattern Matching phases, which would operate much like RSVP [15].

Specifically, the UCP instructs the source to send a "PATH" message to the selected sink node. This message contains the set of end-to-end templates that we are trying to match, as well as a copy of the media and user preference objects. Each node along the route appends its node object to the message. The sink node selects the best template based on the route capabilities it receives, and then sends a "RESV" message along the return path instructing each node what path segment it needs to run.

The remaining question is how many routes the "PATH" message should traverse. At one extreme, the "PATH" message is forwarded along the single shortest route between the source and sink; i.e., each node forwards the message to a single next hop, as in standard IP forwarding. At the other extreme, the "PATH" message is flooded across the network, with each node forwarding the message out all links except the one it arrived on. The flooding process should permit limited cycles so that nodes with excess computational resources are included. Flooding has the benefit of allowing the sink node to select from all possible routes across the network, with the obvious downside of not scaling to large networks. We expect intermediate solutions to be the most likely, where some routers (those in the middle of the network) forward "PATH" messages across a single link, and others (those near the sink node) flood "PATH" messages on all outgoing links.

In summary, the important thing to take away from this discussion is that the number of source-to-sink routes taken into consideration—one, all, k-shortest—bounds the search space, but is otherwise orthogonal to the matching process.

4 Example

This section presents two examples that illustrates how our framework is used to construct network services. The examples revolve around accessing an MPEG-encoded video object.

4.1 Reservation-Based Service

Our first example is not very creative in the code that gets loaded onto the intermediate nodes, but it does attempt to establish an end-to-end path that meets the bandwidth requirements of the media object. In a sense, it is our architecture's equivalent of setting up an RSVP-style flow.

First, assuming the video requires 1.5Mbps of network bandwidth, the corresponding MediaObject is defined as follows:

MediaObject:

```
MediaObject m;
m.bandwidth = 1500;
```

Second, we define a PathRule that binds an MPEG video to the following Template:

{mpg_src ip_fwd* mpg_sink}

where the three NodeRules named in this template are defined as follows:

mpg_src:

```
class mpg_src extends NodeRule
{
    boolean match(Node n, MediaObject m, ...)
    {
        return(m.loc.nid.equals(n.nid)
            && n.hasDevice(storage)
            && n.hasDevice(net));
    }
    NodeTemplate[] templates(Node node)
    {
        // storage/fs/rtp/udp/ip/net
    }
}
```

ip_fwd:

```
class ip_fwd extends NodeRule
{
    boolean match(Node n, MediaObject m,...)
    {
        return(n.hasDevice(net_in)
            && (n.hasDevice(net_out)
            && n.bandwidth()≥ m.bandwidth());
    }
    NodeTemplate[] templates(Node n)
    {
        // net_in/ip/net_out
    }
}
```

mpg_sink:

```
class mpg_sink extends NodeRule
{
    boolean match(Node n, MediaObject m,...)
    {
        return(n.hasDevice(net)
            && n.hasDevice(audio_out)
            && n.hasDevice(video_out)
            && (n.cycles() ≥ cycles(m)));
    }
    NodeTemplate[] templates(Node node)
    {
        // (1) net/ip/udp/rtp/mpeg/video_out
        // (2) net/ip/udp/rtp/mp3/audio_out
    }
}
```

For a given MediaObject m, the mpg_src rule picks the node that has NodeId identical to m.loc.nid, as well as virtual devices storage and net. The mpg_sink rule select the set of sink nodes that have the virtual devices net, video_out and audio_out. The query may include other requirements, such as whether or not the node has enough CPU cycles to decode the video stream, and if the node complies with the attributes from the user's profile UserPref (e.g., a preferred display resolution and location). The user is presented with the resulting set of sink nodes, and selects one of them.

At this stage, the UCP knows the endpoints. The next step is to consult the node database to learn the possible paths between these endpoints; the UCP receives Node objects of all the nodes along the k shortest paths. It then compares these Nodes against the NodeRules using the method outlined in Section 3.2. In our example, zero or more instances of the rule ip_fwd come into play. If there exists enough guaranteed bandwidth on a given node—i.e., if the condition (n.bandwidth()≥ m.bandwidth()) holds—the node is considered to be part of the path. This process continues until the whole node sequence is either accepted by the Template, or rejected.

Once the UCP knows all the feasible NodeRules that are to be installed along the route, we can instantiate the corresponding Node_Template on each of these nodes. For example, two paths—net/ip/udp/rtp/mpeg/video_out and net/ip/udp/rtp/mp3/audio_out—are started on the sink node. Figure 2 shows the resulting path when a sequence of five Nodes matches the Template.

Note that this example is overly simplistic in that the PathRule contains only a single template, and the ip_fwd rule matches only if the bandwidth required by the media object can be reserved. A more realistic end-to-end rule might have included a second template that accepts a router that offers only best effort service, but installs a smart dropping module on the router to deal with congestion that might occur [8].

4.2 Transcoding

Our second example demonstrates how alternative code modules can be installed along the route. To motivate such a scenario, suppose the user is trying to display the video on a PDA across a 1Mbps wireless link. To account for this possibility, the following Template might be returned:

{mpg_src ip_fwd* xcode ip_fwd* mpg_sink}

where the node rule xcode expands as follows:

xcode:

```
class xcode extends NodeRule
{
    boolean match(Node n,MediaObject m,...)
    {
        return(n.hasDevice(net_in)
```

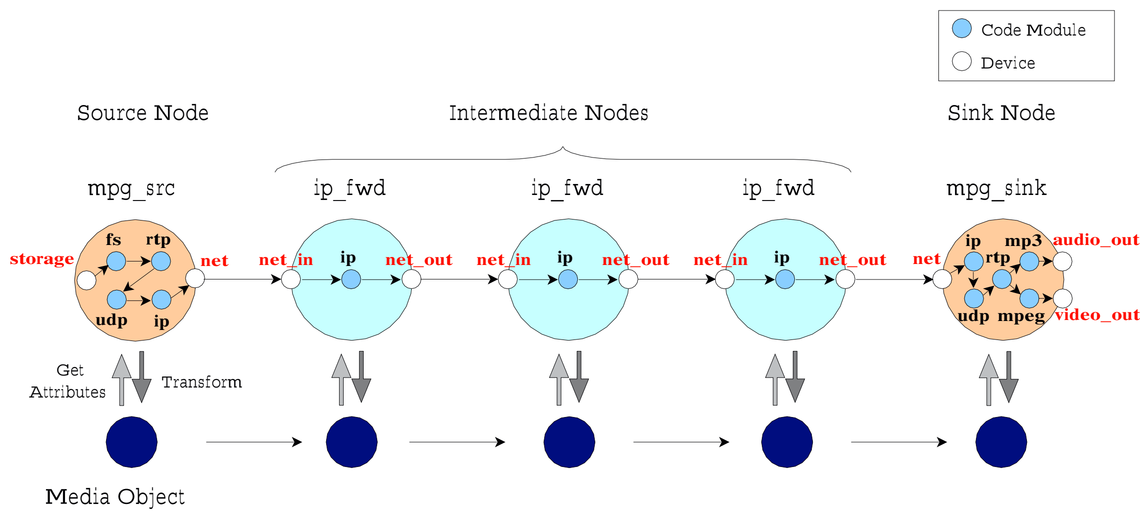

Figure 2: Playing MPEG file with reservation

```
        && (n.hasDevice(net_out)
        && (n.cycles() ≥ cycles_xcode(m)));
    }
    void transform(MediaObject m, UserPref p)
    {
        m=transcode(m,p);
        // this changes the attributes of m
    }
    NodeTemplate[] templates(Node n)
    {
        // net_in/ip/udp/xcode/udp/ip/net_out
    }
}
```

The way to interpret xcode is that, for the cost of cycles_xcode(m), say 50Mcps, it can transcode the video to a lower resolution (m = transcode(m,up))). As a result, the new m requires lower bandwidth, say 1Mbps. The ip_fwd after xcode is applied is identical to the ip_fwd before xcode is applied, except that the former requires higher bandwidth, whereas the latter requires lower bandwidth.

This Template matches any route that has greater than 1.5Mbps available up to some node, and then at least 1Mbps afterwards, as long as one node before the bottleneck has enough cycles to run the transcoder. The matching process identifies the node on which the transcoder will run, and as long as the method xcode.transform() is able to correctly derive the resulting attributes of MediaObject, our framework is able to match the right route. Figure 3 shows the resulting path.

5 Prototype Implementation

This section briefly describes a prototype implementation of our framework, and reports our initial experiences.

5.1 Global Path Construction

Our prototype path resolver—the UCP, the pattern matching algorithm, and the object repositories—are implemented in Java. This was actually our second implementation; the first version used XML [13] to represent the attributes of the various entities in the system. The problem with XML, however, is that it provided no support for modifying attributes during the resolution process, and it limited our ability to consider multiple attributes (e.g., from the node, media, and user preference objects) in the matching algorithm. In the end, we decided that a general-purpose programming language like Java was more appropriate than writing XML scripts to munge attributes. We also foresee the value of Java when we extend our work to use a decentralized algoriothm. We envision that the pattern matcher will be written as a loadable module that runs on a sequence of nodes.

We typically run the UCP on a laptop, but the interface is designed to be suitable for a PDA-sized display. The implementation assumes the full JVM, which may be more than a realistic PDA can afford. We need to explore subsets of the JVM designed for embedded systems. On the other hand, this problem is related to our decision to treat the UCP as the central point of control. If we had off-loaded this work to various nodes in the network, then the UCP could be made much thinner.

Our pattern matching algorithm is currently restricted to simple regular expressions. While it is powerful enough to formulate the rules for a sequece of nodes, we would like to explore the way to match a tree of nodes. This gives much more versatility to our scheme, for example, in establishing a multicasting path for video conferencing.

As mentioned previously, our pattern matching is non-deterministic, which clearly causes a scaling problem. Currently, we do exhaustive search for all the node rule sequences possible by backtracking non-deterministic branches, which requires copy of media objects and node rule sequences. We did not have complete solution to reducing the complexity. This is one of our future works. We

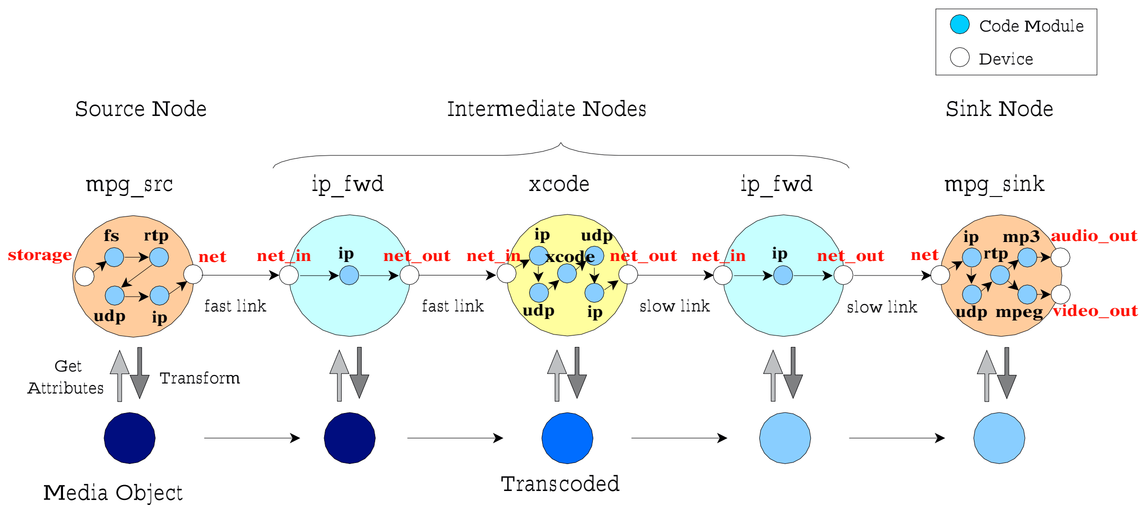

Figure 3: Playing MPEG file with transcoding

may need to make a node rule as specific as possible or to utilize network overlay to virtually decrease the size of network.

5.2 Local Path Construction

Each node in the system runs Scout, which we have extended in two ways. First, we have added a path manager, called Ranger, that accepts the NodeTemplate contained in the node rule. Ranger parses the template, resolves any virtual devices in the template, downloads any missing modules, and invokes Scout's PathCreate operation to instantiate the path. Prior to Ranger, path creation happened incrementally, with each module selecting the next module on the path based on attributes (invariants) provided as input. The new solution places the decision as to what sequence of modules constitute a path in the hands of the "path programmer", who stores the valid paths as rules in the path database.

Second, we have added a code loader/linker, called KMOD, to Scout. Earlier versions of Scout were limited to the set of modules configured into the system at build time. It is now the case that Scout is built with only the loader path shown in Figure 4 pre-installed. KMOD accepts modules in the ELF format, and then patches the kernel symbols. Table 5.2 reports the size of kernel corresponding to the path defined in object mpeg_sink. The column labeled ELF gives the size of the .o file for the module; the "Loaded" gives the size of the sections actually loaded into Scout. The base kernel size (text only) is 296.1KB, so a Scout kernel with the MPEG path dynamically loaded occupies 457.8KB. By comparison, a statically configured kernel with the same modules has a text size of 402.8KB.

The KMOD load/link method is not very sophisticated. An object file in ELF format (.o file) tends to be much bigger than the text that gets loaded in. Also, each module must be compiled with the kernel to share exported symbols.

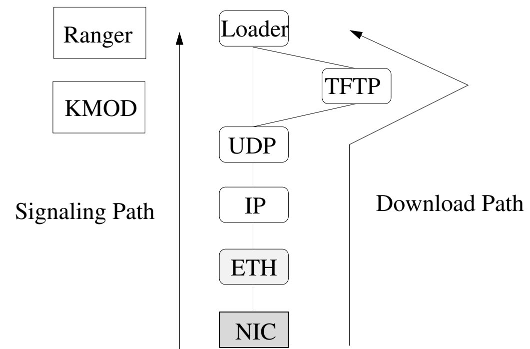

Figure 4: Loader Path

Modules	ELF	Loaded
MPEG	587KB	120.4KB
Mflow	127KB	11.1 KB
WIMP	51KB	4.7KB
Shell(Ctrl)	48KB	5.5KB
S3	262KB	13.8KB
VGA	69KB	6.2KB

Table 1: Size of modules on sink node for MPEG video path

6 Related Work

The work described in this paper combines elements of resource discovery, active networking, and QoS routing, all of which have been studied in other settings.

Sun's *Jini* [12] and Microsoft's *Universal Plug and Play (UPnP)* [10] support resource discovery in much the same way as we do. Jini's architecture is similar to ours in that there is a centralized *Lookup Server*, analogous to our object repositories. Each service is responsible for finding and registering with some Lookup Server, similar to a node registering itself with the central node database in our prototype. The client locates a service by Java serialized object type (or subtype), along with descriptive attributes of

the service. Although at a high level our system has the same goal as Jini, we also considers the functionality of intermediate nodes, which are largely ignored in the Jini architecture.

UPnP is an architecture for pervasive peer-to-peer network connectivity of devices. Contrary to Jini and our framework, UPnP does not have centralized entity to manage resources. Instead, each device implements a collection of servers—*Discovery*, *Description*, *Presentation*, *Control* and *Events*—to answer requests from a client. There are six steps in establishing a connection between a client and a device. After the client and devices get each other's addresses (*Addressing*), the client finds an interesting device (*Discovery*), learns about the device's capabilities (*Description*), invokes actions on the device (*Control*), listens to state changes of the device (*Events*), and controls the device using an HTML-based user interface (*Presentation*). The Description step leverages XML to describe services available on devices. Although the goal of UPnP is similar to our goal, the architecture is different in many details, and most importantly, UPnP is limited to direct connection between two end devices, with no intermediate nodes explicitly involved in the connection.

In terms of active networking, our work can be viewed as defining a signalling protocol for installing functionality on the intermediate nodes (routers) in a network. Our framework also defines a method of deploying modules necessary for implementing a new protocol along an end-to-end path, and so it is similar to execution environments like ANTS [14]. One key difference is that ANTS uses a decentralized strategy, while we currently employ centralized decision making. In addition, our framework considers resource allocation issues, whereas ANTS is purely a best effort system.

A slightly different approach is taken by Active Services [2], which implements a Media Gateway with functionality similar to our transcoder (xcode) example. Our approach is more general in that it encompasses both transcoders and finer-grained IP extensions, and we allow for more degrees of freedom in selecting the appropriate node to run the service on.

Our work also includes elements of QoS routing, which addresses two problems: collecting up-to-date state of an entire network, and selecting a feasible path between source node and sink node based on this information [4]. In contrast to approaches that hierarchically aggregate global state, and hence focus on approximations using imprecise information, our architecture assumes a simple strategy in which each node maintains its own state, and the node database tracks the global state of the network.

Ninja Paths [3, 6, 1] are probably the most related project to our framework. Ninja introduces *operators* as a program to perform the simplest service. The *Automatic Path Creator (APC)* generates a *logical path*, or a sequence of compatible named operators, to compose a complex service out of many simple services, which are then *instantiated* into a *physical path* by the *Service Discovery Service (SDS)* [5]. Once a physical path is set up, the constituent operators are started (*implemented*). The *uploadable operator* could be uploaded onto the machine running the *source operator* to decrease network traffic. However, the current implementation of Ninja Paths assumes operators must already be running at the construction of the logical path. We see much similarity between Ninja Paths and our framework: our per-node modules, Template, and sequence of Scout nodes, correspond to Ninja's operators, logical path, and physical path, respectively. There are many differences, however:

- The generation of logical paths by APC lacks semantic controls and relies on the correctness of operator connectivity and optimization of its path length, while our Template clearly defines the semantics of entire path. In other words, path semantics are controlled by programmers in our system.
- The instantiation of logical paths is based on random selection among qualified actual operators. In contrast, we use pattern matching based on attributes of intervening entities, such as routers and links.
- The granularity of path construction is only at the operator level, which corresponds to the node level in our framework. In contrast, our approach extends to the device/module level, thereby providing finer grained control.
- No control is available over multiple flows going through one host, whereas each node running Scout can schedule multiple flows in our framework.
- Operators are usually assumed to be either already running or only uploadable onto the source node, while our system allows each node to download modules and device drivers on demand.

In summary, Ninja paths put more stress on composability of basic services that are already available, while our focus is on data-centric semantics control for media object delivery. Computer-Aided composition of services might help programmers write correct Template's, but we would separate that process from constructing end-to-end paths.

7 Conclusions

This paper describes an architecture for establishing paths over which media objects can be accessed. Each end-to-end media path consists of (1) a route through the network that connects a source device to a sink device, (2) the code on each member node needed to transform and forward the object, and (3) the resources (e.g., cycles, bandwidth) needed to play the object.

In describing an architecture for end-to-end paths, the paper makes two contributions. The first is to define the relationship between the process of discovering a suitable route through the network, and the process of determining what code should run on each node along that route. Rather than fixing the code that runs on each router and discovering the route with the best QoS properties, or alternatively, settling for the shortest route and then selecting the appropriate code to run on each router, *our approach integrates route selection and code selection into a single process.* The second contribution is to explicitly support the four entities that influence route/code (path) selection: the resource requirements of the media object, the user's preferences, the capabilities of each node in the network, and the programmers that define the available services.

Although we have a prototype implementation of the architecture, we do not yet have sufficient experience with the system to comment on its robustness, usability, and programmability. In particular, we recognize that the current centralized resolution mechanism limits the size of network our system can accommodate, and so our plan is to further refine and implement the distributed approach outlined in Section 3.2.3. We also plan to implement a wider range of services and media objects.

References

[1] Ninja Project. Computer Science Division, University of California, Berkeley, http://ninja.cs.berkeley.edu/.

[2] E. Amir, S. McCanne, and R. Katz. An Active Service Framework and its Application to Real-time Multimedia Transcoding. In *Proceedings of the ACM SIGCOMM '98 Conference*, pages 178–189, August–September 1998.

[3] S. Chandrasekaran, S. Madden, and M. Ionescu. Ninja Paths: An Architecture for Composing Services Over Wide Area Networks. http://ninja.cs.berkeley.edu/pubs.html.

[4] S. Chen and K. Nahrstedt. An Overview of Quality-of-Service Routing for the Next Generation High-Speed Networks; Problems and Solutions. *IEEE Network Magazine, Special Issue on Transmission and Distribution of Digital Video*, 12(6):64–79, November–December 1998.

[5] S. E. Czerwinski, B. Y. Zhao, T. D. Hodes, A. D. Joseph, and R. H. Katz. An Architecture for a Secure Service Discovery Service. In *Fifth Annual International Conference on Mobile Computing and Networks*, pages 24–35, August 1999.

[6] S. D. Gribble, M. Welsh, R. von Behren, E. A. Brewer, D. Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R. Katz, Z. Mao, S. Ross, and B. Zhao. The Ninja Architecture for Robust Internet-Scale Systems and Services. To appear in a Special Issue of Computer Networks on Pervasive Computing.

[7] A. Hopper. Sentient Computing. University of Cambridge and AT&T Laboratories Cambridge Ltd. http://www.uk.research.att.com/abstracts.html.

[8] R. Keller, S. Choi, M. Dasen, D. Decasper, G. Fankhauser, and B. Plattner. An Active Router Architecture for Multicast Video Distribution. In *IEEE INFOCOM 2000*, March 2000.

[9] S.-W. Lee and C.-S. Wu. A k-Best Paths Algorithm for Highly Reliable Communication Networks. *IEICE Transactions on Communications*, E82–B(4):586–590, April 1999.

[10] Microsoft. *Universal Plug and Play Device Architecture Version 1.0*, June 2000. http://www.upnp.org/UPnPDevice_Architecture_1.0.htm.

[11] D. Mosberger and L. L. Peterson. Making Paths Explicit in the Scout Operating System. In *Proceedings of the Second USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 153–167, October 1996.

[12] Sun Microsystems. *Jini Technology Architectural Overview*, January 1999. http://www.sun.com/jini/whitepapers/architecture.html.

[13] W3C. *Extensible Markup Language(XML)*. http://www.w3c.org/XML.

[14] D. Wetherall. Active network vision and reality: lessons from a capsule-based system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 64–79, December 1999.

[15] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A New Resource ReSerVation Protocol. *IEEE Network Magazine*, 7(5):8–18, September 1993.