Report Number: WUCS-00-27

2000-01-01

Programming Active Networks Using Active Pipes

Ralph Keller, Jeyashankher Ramamirtham, Tilman Wolf, and Bernhard Plattner

Active networks allow customized processing of data traffic within the network which can be used by applications to improve the quality of their sessions. To simplify development of active applications in a heterogeneous environment, we propose active network pipes as a programming abstraction to specify transmission and processing requirements. We describe a routing algorithm that maps application session requirements onto network resources and determines an optimal route through the network transiting all required processing sites. Additionally, we propose a network software architecture to implement the functionality required to support active pipes.
... Read complete abstract on page 2.

Programming Active Networks Using Active Pipes

Ralph Keller, Jeyashankher Ramamirtham, Tilman Wolf, and Bernhard Plattner

Complete Abstract:

Active networks allow customized processing of data traffic within the network which can be used by applications to improve the quality of their sessions. To simplify development of active applications in a heterogeneous environment, we propose active network pipes as a programming abstraction to specify transmission and processing requirements. We describe a routing algorithm that maps application session requirements onto network resources and determines an optimal route through the network transiting all required processing sites. Additionally, we propose a network software architecture to implement the functionality required to support active pipes.

Programming Active Networks Using Active Pipes

Ralph Keller, Jeyashankher Ramamirtham,
Tilman Wolf and Bernhard Plattner

WUCS-00-27

October 2000

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
St. Louis MO 63130

Programming Active Networks Using Active Pipes

Ralph Keller[1], Jeyashankher Ramamirtham[2], Tilman Wolf[2], Bernhard Plattner[1]
[1] [keller | plattner]@tik.ee.ethz.ch [2] [jai | wolf]@arl.wustl.edu
[1] Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology,
Gloriastrasse 35, 8092 Zurich, Switzerland, +41-1-632-7015
[2] Department of Computer Science, One Brookings Drive, Campus Box 1045,
Washington University, St. Louis MO, USA, +1-314-935-4845

Abstract — **Active networks allow customized processing of data traffic within the network which can be used by applications to improve the quality of their sessions. To simplify development of active applications in a heterogeneous environment, we propose *active network pipes* as a programming abstraction to specify transmission and processing requirements. We describe a routing algorithm that maps application session requirements onto network resources and determines an optimal route through the network transiting all required processing sites. Additionally, we propose a network software architecture to implement the functionality required to support active pipes.**

Keywords — **Active networks, active application programming, session configuration, constraint-based routing**

I. Introduction

Active networks provide processing capabilities on routers that allow customized handling of data traffic within the network [8]. Processing in the network can be used to deploy new services, improve end-to-end quality, monitoring, and many other useful applications. The main challenge of providing secure, reliable, high performance, and programmable execution environments on routers has been addressed by many research groups [11], [2].

In most cases, it is difficult for application programmers to make use of active services in the network since it requires an understanding of the underlying network infrastructure and the system architecture of the active router. In this paper, we propose a method by which these details can be hidden from the programmer, making the development of the application simpler. In particular, we address the following three issues:

- We introduce a method for specifying transmission and processing requirements for connections over active networks.
- We demonstrate how the connection requirements can be mapped onto network resources while minimizing network costs.
- We propose a network software architecture that can implement the services required for resource mapping, routing, and connection setup.

The basic idea of specifying transmission and processing requirements is to model a connection as a sequence of functions that have to be performed on the data stream. This concept is analogous to *pipes* in UNIX where data can be sent through a sequence of programs. In the active networking environment, each function corresponds to a *code module* that has to be installed on a router along the path of the connection. An *active pipe* is a more general definition of an execution sequence since processing requirements can be optional and can depend on the state of the network. We require that the connection always goes through the same sequence of code modules and, therefore, assume connection-oriented sessions.

In Section II, we describe our programming paradigm for active networks and explain how active pipes can be defined. Section III illustrates how we can map an active pipe onto network resources and determine an optimal route and the location of processing sites. In Section IV, we present a network software architecture to implement the required functionality. In Section V, we discuss related work and Section VI concludes this paper by summarizing results.

II. Programming Paradigm for Active Networks

The formal definition of an active pipe, that we introduce in this section, is used by an application to specify session requirements.

A. Active Network Pipes

An active network pipe is a description of a processing sequence used in a connection from the source to the sink over an active network. The active pipe abstraction can be used by the application programmer to define where active code modules need to be deployed for a particular connection. There are two types of code modules that can be used in an active pipe:

- A *required module* provides a processing function that *must* be performed exactly once in the network. This functionality is essential for the correct operation of the application and cannot be omitted. Such a processing function typically changes the format of the data stream (e.g., encryption, media transcoding).
- A *conditional module* provides a functionality that is not necessary for correct operation, but can improve the quality of the connection. This type of code mod-

Pipe = **source** *Modules* **sink** ∨ *Pipe* | **source** *Modules* **sink**
Modules = *Selector* [*Conditionalmodules*] *Requiredmodules Modules* | ε
Selector = **sel**(*S*) | ε
Conditionalmodules = *Conditionalmodule Conditionalmodules* | ε
Requiredmodules = *Requiredmodule* | ε
Conditionalmodule = *Modulename* (*Location, Installationcondition*)
Requiredmodule = *Modulename* (*Location, Installationcondition*)
Location = **node** | **link**
Installationcondition = *C*
Modulename = **encryption** | **congestioncontrol** | **monitoring** | **transcoding** | ...

Figure 1: Active pipe grammar

ule is installed on all nodes that fulfill a given condition. As a result the code module can be deployed multiple times along the path. If the condition is not satisfied at all, then no modules need to be installed. These code modules typically do not change the format of the data stream (e.g., monitoring, congestion control).

In addition to defining a sequence of required and conditional modules, the programmer can use three other components, which can be used independently on all modules, to express the processing demands:

- *Network attributes* that maintain static and dynamic state information for links and nodes,
- *selectors* that restrict the links and nodes that are considered for a connection, and
- *installation conditions* that describe circumstances under which an active code module should be deployed.

1) Network Attributes

In our network model we attach *attributes* to all nodes and links to define properties of nodes and links. Attributes can be either *static* or *dynamic* and are configured by network management. For example the "address" of a node is a static attribute whereas the "available bandwidth" on a link is a dynamic attribute.

Formally, an attribute consists of a two-tuple <*name, value*>. Each node or link can have multiple attributes defined as an *attribute set* $A = \{a_1, a_2, a_3 ... a_n\}$, where each a_i is an individual attribute. We refer to the value of attribute *name* on node *v* as *v.name* and on link *e* as *e.name*.

2) Selectors

In certain cases the path of a connection needs to be restricted to a certain set of nodes and links. For example, packets that are not encrypted might not be allowed to leave a certain domain. For this purpose we define a selector that restricts the graph on which routing is performed. Let the network be represented as a graph $G = (V, E)$ with nodes V and links E. The selector $S: G \rightarrow G'$ maps the complete network graph G to a subset $G' = (V', E')$. All routing decisions are then restricted to that particular graph G'. If the selector is omitted, routing is performed on the complete network graph G.

3) Installation Conditions

Code modules are only installed on nodes and links if the installation condition is fulfilled. For this purpose, we define the installation condition C as a function that maps nodes or links to a boolean value $C: V \rightarrow \{true, false\}$ or $C: E \rightarrow \{true, false\}$, respectively. For the evaluation of this function, node and link attributes are used. Based on this function, we can determine the *qualifying set* of links or nodes, which is the set of all locations where the installation condition is satisfied.

4) Active Pipe Grammar

With the definition of attributes, selectors, and installation conditions, we can express conditional execution of code modules on one or many nodes or links. The grammar shown in Figure 1 defines an active pipe as a sequence of required modules, conditional modules, and selectors. An example of a module is **encryption (node, node.domain = A)**, where **encryption** is the identifier of the module, **node** indicates that it should be installed on a node, and **node.domain = A** is the condition under which it is installed. For the selector *S* and the installation conditions *C* in the grammar, we assume the functions described in the previous subsections.

As an example for an active pipe, we can think of a scenario where a connection for sensitive data transmission is established between two domains. If the two domains are directly connected to each other, traffic does not need to be encrypted but if the traffic transits nodes outside the trusted domains, encryption is needed and monitoring within the untrusted domain on some nodes is desired as illustrated in Figure 2.

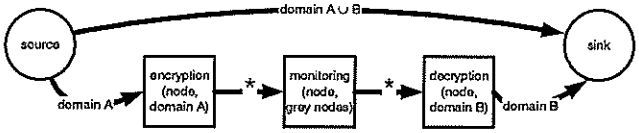

Figure 2: Active pipe for encryption between domains

Selectors are attached to the arrows between code modules and the installation conditions are shown below the module names. Using the grammar, the example can be expressed as:

```
source sel (domain A ∪ domain B) sink ∨
source
  sel (domain A) encryption (node, node.domain = A)
  sel (*) [ monitoring (node, node.type = grey) ]
  decryption (node, node.domain = B) sel (domain B)
sink
```

More detailed explanations of various deployment scenarios are given in the following section.

B. Deployment Scenarios

With the formal definition of an active pipe, we can address the following four basic deployment scenarios:

- required processing on node
- required processing on link
- conditional processing on nodes
- conditional processing on links

For the discussion, we use the graph shown in Figure 3.

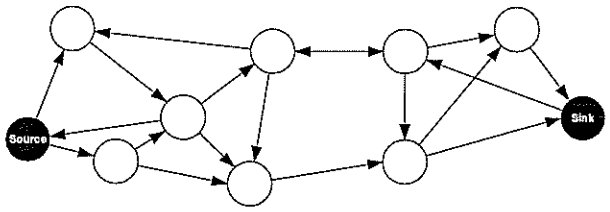

Figure 3: Network graph used for scenarios

1) Required Processing on Node

In this example, processing *on exactly one node* is required. The path of the connection has to be set up such that it traverses at least one node that can perform the required processing. Figure 4 shows those two potential processing nodes in grey and a path that traverses the left processing node. Example for this type of processing is data transcoding. Using the grammar, this can be expressed as:

```
source transcoding (node, node.active = true) sink
```

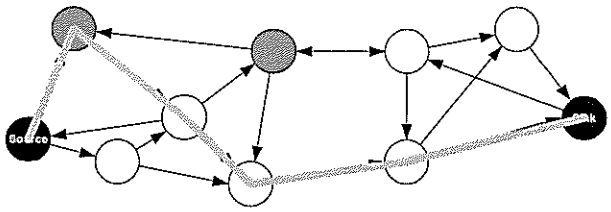

Figure 4: Required processing on a single node

2) Required Processing on Link

Similar to processing on one node, we can require processing *on exactly one link.* In Figure 5, two links are marked where processing could be performed on the nodes that precede the links. This type of processing could be encountered in conjunction with the Differentiated Services architecture that allows network providers to offer services with different quality-of-service to traffic streams. The classification, marking, policing, and shaping operations are implemented at network boundaries and routers within the core of the network handle packets in different traffic classes according to the per-hop behaviors indicated in the packet header. The links in Figure 5, for example, could be preceded by traffic shaping modules. This can be expressed as an active pipe as:

```
source trafficshaping (link, link.leavesdomain = true) sink
```

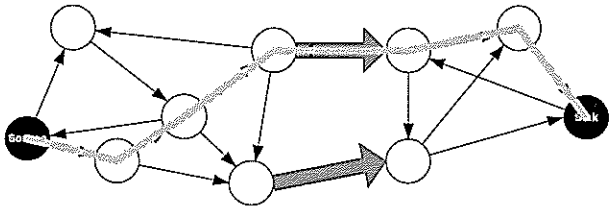

Figure 5: Required processing on a single link

3) Conditional Processing on Nodes

When processing is conditional, it can happen arbitrarily many times (including not at all). If the path traverses a node which qualifies for processing under the given condition, the code module will be installed. One example for this type of processing is traffic monitoring on routers that are of type *M*. Figure 6, depicts nodes on which monitoring can be performed as shaded. The final path traverses two such nodes and the monitoring module would be installed on both of them. As an active pipe, this can be expressed as:

```
source [ monitoring (node, node.routertype = M) ] sink
```

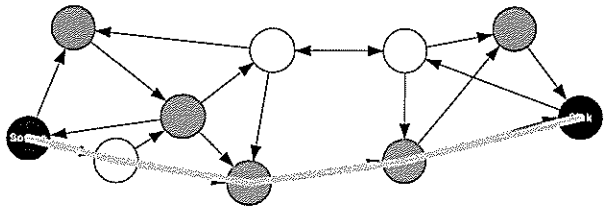

Figure 6: Conditional processing on nodes

4) Conditional Processing on Links

Similar to conditional processing on nodes, processing can also be performed on links. A good example for such a scenario is application-specific congestion control. The application can install customized modules that modify the application's data stream in response to congestion on certain links. For example video congestion control modules in the network can minimize the impact of losses on the perceived video by preferentially discarding high frequency image coefficients from the video stream as demonstrated in [10]. If the data stream is routed along several congested links, then congestion control modules should be installed at each of these links. If the stream is not routed through any congested link, no congestion control modules need to be installed. Figure 7 illustrates links where congestion control could be performed with large arrows. The active pipe expression for this scenario is:

source [congestioncontrol (link, link.congested = true)] sink

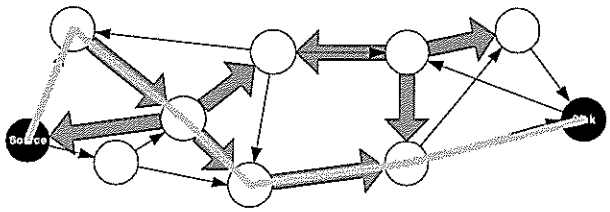

Figure 7: Conditional processing on links

Based on the formalism for defining transmission and processing steps in a connection over an active network, we will now demonstrate how resources are mapped onto the underlying network and how an optimal route and the location of processing sites can be determined.

III. Constraint-based Routing in Active Networks

Routing in active networks has to find a least-cost path from the source to the destination taking into account transmission costs over links and costs of executing modules. We assume that processing cost at an active node is determined by the profile of the active module (e.g., required processing cycles, memory usage) and is scaled to match the link cost units. This is a convenience that allows us to transform the routing problem into a simple shortest path problem with just link costs. Moreover, shortest path problems with more than one cost metric are known to be intractable [9].

We describe how the routing is performed for a single module case (*required or conditional)* and then we look at how the routing works for a sequence of active modules in an *active pipe*. The main idea is to transform all the cases into a shortest path problem and use the methods incrementally to generate a shortest path problem for the *active pipe* case.

Formally, the problem for the single module case can be stated as follows. We are given a directed graph, $G = (V, E)$, with a transmission cost $c(e)$, for each link $e \in E$, and a processing cost of the module $c(v)$, for each node $v \in V$. Let the source be defined by s and the destination by d. We describe the transformations for each of the scenarios described in Section II to a shortest path problem. As an example network, we use the graph shown in Figure 8. Transmission costs are denoted on the links and the processing costs for a module are shown within the nodes.

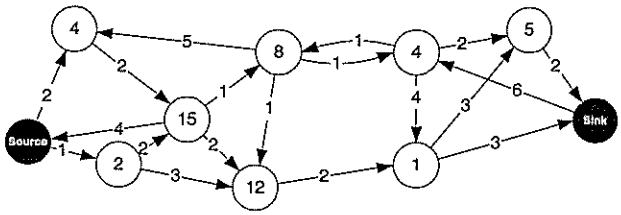

Figure 8: Network graph used for scenarios

A. Required Modules

For required modules, the routing algorithm's objective is to determine the node where processing can be performed such that the cost of the path (sum of transmission costs on links and the processing cost for the active module at the node) is minimized. As discussed in Section II, there are two cases:

1) Required Processing on Nodes

In this case, given a *qualifying set* of nodes, $N \subseteq V$, we need to determine the optimal node $n \in N$, where the processing should be done. As described in [6], we can solve this problem by transforming it to a shortest path problem. We modify the graph G by making two copies which we identify as layer 1 and layer 2 as illustrated in Figure 9. For each vertex u in the initial graph, let u_1 denote the vertex in layer 1 of the target graph while u_2 denotes the vertex copy in layer 2. To model the processing of modules, we add edges *between* the two layers. For every node $n \in N$, where processing may occur, we add an edge

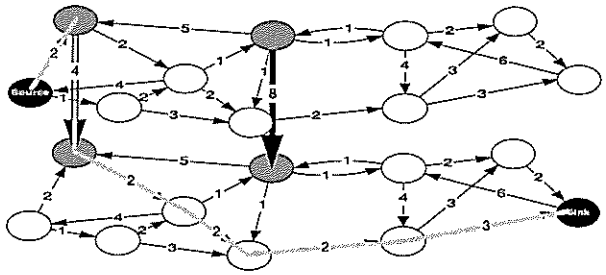

Figure 9: Transformation for required node scenario

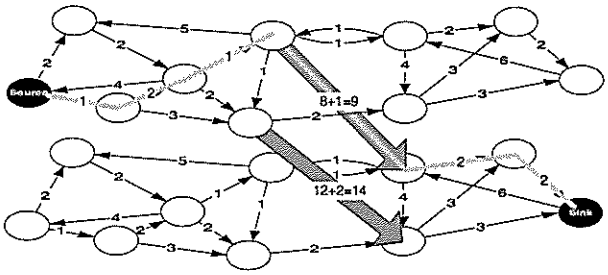

Figure 10: Graph transformation for required link scenario

(n_1, n_2) in the target graph and let the link cost of (n_1, n_2) be the processing cost on node n, $c(n)$.

The source node in layer 1, s_1, is the source for this new graph and the destination node in layer 2, d_2, is the destination node for the new graph. This ensures that the path from the source to the destination is forced to go through exactly one processing site. To resolve the routing problem with one mandatory processing site, we find a least-cost path in the target graph using a shortest path algorithm. The path can be mapped back to the original graph by projecting the two layers onto a single layer and the processing is optimally performed where the path crosses the two layers. A proof for this method is presented in [6].

2) Required Processing on Links

Here, given a *qualifying set* of links, $L \subseteq E$, the application wants the processing to be done *on the node adjacent to exactly one link* among the set. In this case, the procedure of determining the optimal location is very similar to the previous case, except for one small variation. Again, we transform the graph G by making two copies, as in the previous case as illustrated in Figure 10. For each vertex u in the initial graph, u_1 denotes the vertex in layer 1 of the target graph while u_2 corresponds to the vertex copy in layer 2. Now for every edge $e \in L$, which connects nodes i and j, we add a new *diagonal edge* (i_1, j_2) in the target graph *between* the two layers. The weight of this new edge is the sum of processing cost at the node and the transmission cost of the link. This is given by the expression $c(i) + c(e)$. The shortest path from s_1 to d_2 gives us an optimal path that transits the processing site and the link crossing the two layers is the optimal location to do processing.

The proof of this is straight-forward. To reach the destination d_2 from s_1, we need to traverse one of the edges connecting the two layers. The edge weights of these links ensure that when we cross a layer, we take into account the cost of installing the active module before the link. Since the shortest path algorithm selects a route with minimum costs, the edge that crosses the layers is the optimal location to install the active module.

Note: An important fact to observe about routing in the *required* processing cases is that *all paths* from *any* node in layer 1 to *any* node in layer 2 satisfy the constraint that an active module must be installed on a *single node* of the qualifying set. In other words, to reach any node in layer 2, we have to traverse one of the links connecting the two layers. No matter how we modify layer 2 to do other processing, any path from layer 1 must use one of the links connecting the two layers. Thus, we can modify the graph in the lower layers and the constraint that an active module must be installed at one location is still satisfied in the resulting shortest path problem through this transformation. This shows the correctness of the shortest path problem that is constructed for the *required* modules.

B. Conditional Processing

In the conditional processing case, active modules should be installed at locations (with respect to nodes or links) that satisfy a given condition along the path. We now discuss the two cases presented in Section II:

1) Conditional Processing on Nodes

In this case, we are given a qualifying set of nodes, $N \subseteq V$, and we need to determine the set of nodes where the active module must be installed such that the path cost is optimal. More formally, suppose there exists a path $p = <s, v_1, v_2 ...., d>$ from the source s to the destination d, and if $v_i \in N$, then an active module must be installed on that node v_i. This can be solved as follows. For every node $v \in N$, let $\{e_1, e_2, e_3, ... e_n\}$ be the set of outgoing edges. As shown in Figure 11, the transformation on the graph is simply to increase the edge weights of the outgoing links by the processing cost of the node. That is, $c_{new}(e_i) = c(e_i) + c(v)$. The source and destination remain the same.

Proof of this method is, again, straight-forward. When a path transits a node, the processing costs at the node are taken into account by the increased link weight of the outgoing link. Thus, when a path goes through a node, an active module can be deployed at that node. Since the shortest path algorithm optimizes total costs, both link

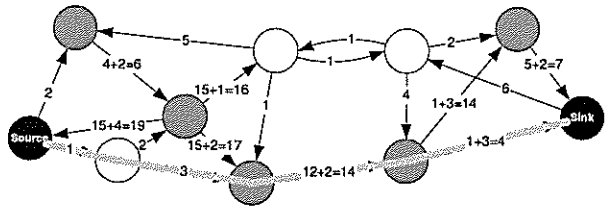

Figure 11: Transformation for conditional nodes scenario

and processing costs are minimized.

2) Conditional Processing on Links

Here, the application wants a module to be installed at all links in the path given a *qualifying set of links*, $L \subseteq E$. This case is similar to the node case, except that the link weights are increased corresponding to the links in the *qualifying set* only. Thus, the transformation as illustrated in Figure 12 on the graph is, for every edge $e \in L$, increase the edge weight by the processing cost of the node adjacent to that link. That is, $c_{new}(e) = c(e) + c(v)$, where e is an outgoing link at v.

The proof directly follows from the proof of the previous case with the difference that the active module needs to be installed if we go through the particular links only. Thus, the increase in the edge weights of these links ensures that if any of the links is taken, then the cost includes the processing cost too.

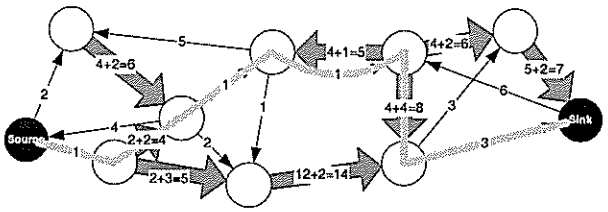

Figure 12: Transformation for conditional links

Note: In the conditional cases as well, the important fact to notice is that *all* paths from *any* source to *any* destination satisfy the given condition that if the path transits any node (or link) that belongs to the qualifying set, then an active module is installed there. This is true because the link weights are increased in such a way that the processing cost of the active modules is included in the link costs. Thus, the shortest path problem includes the cost of installing the active modules.

C. Combination of Conditional Modules

In the specification of an active pipe, a *combination of conditional modules* can be installed on the same graph and the order of installation does not matter. The graph, G, is transformed using the method described for the *conditional module* case, individually, for each of the modules on the same graph. This results in a graph with link weights modified for each of the conditional modules. For each of the conditional modules, all paths between any two nodes in the graph satisfy the required constraint that "whenever going through a selected node or link, the corresponding active module needs to be installed". This follows directly from the observation mentioned about the conditional module transformations above. Also, if the installation condition for two different modules overlap (i.e., some node or link is a candidate for installing the active module for both cases), then the flow has to be processed by both modules at the same location. We assume that active nodes support chaining of active modules for a single flow, i.e., a single flow can be processed by multiple code modules in a sequence on the same node.

D. Active Pipe Routing

We shall now define an algorithm to perform routing in the *active pipe* case. An *active pipe* can have various alternatives that are separated by '∨'. The different alternatives are solved, *individually*, for the shortest path and the optimal one is chosen among them. So, we discuss how the routing is done for a single alternative. As described in Section II, a *selector* restricts the route until some processing is done. The selector defines a subgraph through which the routes can be taken until a required processing is performed or the destination is reached. For each *required module*, we add a new layer in the graph according to the transformations defined previously. The second layer is defined by the next selector specified by the application and if there is none, the first layer is replicated to give us the second layer. A number of *conditional modules* can be installed together *on the same layer* according to the specification by the application. This results in modifying link weights in the layers for each of the modules. The original source node in the topmost layer of the graph is the source of this modified graph and the original destination node in the bottommost layer is the destination of the modified graph. The resulting graph satisfies the constraints specified by the application because the constraints are satisfied for each of the modules. Solving the resulting graph for the shortest path returns optimal locations for installing the modules.

As an example, we consider the active pipe from Section II. The first alternative is to determine a path that does not leave the trusted domain and in our example there does not exist such a path. Instead, we need to go through an encryption and decryption step. Figure 13 shows the graph obtained by applying the transformations for this case. The top layer is the source domain defined by the selector **sel (domain A)**. The required

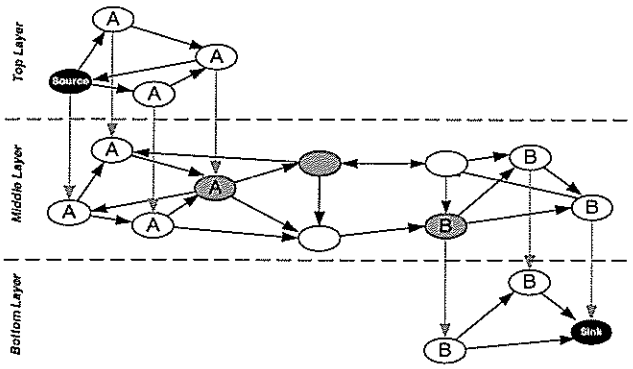

Figure 13: Graph transformation for encryption and decryption example

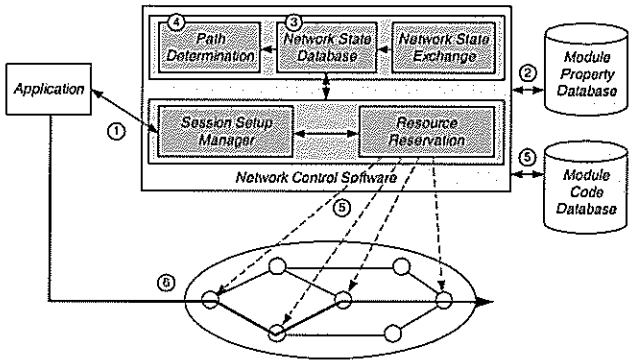

Figure 14: Network Control Software (NCS)

encryption step is done between the top and the middle layer and the required decryption is performed between the middle and the bottom layer. The conditional monitoring is done in the middle layer at the grey nodes. Using a shortest path algorithm, we can now determine the optimal route and processing locations.

IV. Network Software Architecture

We have described an algorithm that allows us to translate application requirements onto network resources. In this section, we identify the required components needed for an implementation and propose an architecture that obtains and maintains network state information required for optimal resource mapping. Additionally, the software is responsible for installing code modules at the locations determined by the routing algorithm.

A. Network Control Software

The core component of our network architecture is the *Network Control Software* (NCS), a distributed session configuration system for flows that need processing on intermediary node. The NCS accepts session initiation requests from applications, maps the session requirements onto network resources, and reserves resources along the least-cost path. Each domain runs an instance of the NCS which has complete information about its local domain and aggregated information on distant domains. If a session needs to be configured using multiple domains, the local NCS computes a path based on its view of the network, reserves processing resources in its domain, and then forwards the session configuration request to successive instances of the NCS along the path to the destination.

Internally, the NCS is composed of the following components as depicted in Figure 14:

- The **session setup manager** handles session initiation requests from the application and controls other components for session setup.
- The **network state database** stores the current state of the network which includes the topology and the location and work-load of processing elements.
- The **network state exchange** component continuously updates the network state database with information reported by individual nodes.
- The **module property database** contains module descriptions for use by the NCS. Specifically, it contains information such as whether the module should be installed with respect to a node or link, and whether it is required or conditional. In addition, the database includes the installation condition and the selector functions. Storing module-specific properties in a database simplifies the API, because the user does not need to deal with module-specific parameters.
- The **module code database** provides a library of active modules for (possibly) different type of execution environments.
- The **path determination** component performs constraint-based routing by internally building the layered graph and then using a shortest-path algorithm computes a least-cost route and the location of processing modules.
- The **resource reservation** component allocates network resources along the session path. Using a protocol such as [5], active modules are installed in the router's networking subsystem.

B. Module Properties

A description of each module is stored in the *module property database* and includes information such as whether the module should be installed with respect to a *node* or *link*, and whether it is *required* or *conditional*. Each module also contains a list of formal parameters that are needed to instantiate a module (such as the "trusted domain" for the encryption module). To specify the eligible locations where the module can be installed,

the description contains the installation condition. Table 1 illustrates the module description for an encryption module. Note that the function uses the formal parameter *trusteddomain* which is provided by the application.

Table 1 Properties for encryption module

Encryption Module	
formal parameters	*trusteddomain*
module type	required
location	node
installation condition	if *v*.domain = *trusteddomain*
selector	*trusteddomain*

To install a module in the network, the application has to provide only the name of the module and its formal parameters. All module-specific information (such as the module type, installation condition, selector) is retrieved from the module property database. For example, an application that wishes to install encryption and decryption modules needs to provide the following information:

```
<encryptionmodule, "arl.wustl.edu">
<decryptionmodule, "tik.ee.ethz.ch">
```

Using the information provided by the application and the information stored in the property database, the NCS can build the graph and determine a route for the session as described in Section III.

C. Session Setup

Now, we describe the session setup procedure using the encryption example. The following six session setup steps are illustrated in Figure 14:

1. Using an active pipe description, the application requests a secure session between two domains. The domain identifiers are passed as formal parameters.
2. The session manager retrieves information about the encryption and decryption modules from the module description database.
3. The network state database delivers information about the topology and state of the network.
4. Using the network state information, the routing component determines a path through the network and the location of active modules along that path.
5. The resource reservation component allocates the required resources and installs the code modules along the path.
6. Once all active modules are deployed and the connection is setup, the application can start transmitting data.

D. Implementation Issues

For an implementation of our proposed network architecture, routers are required to support connection-oriented communication. This can be achieved by either using routers that inherently support connection-oriented communication, like ATM switches, or by building a connection-oriented overlay network over a datagram network, similar to Virtual Private Networks used in the Internet.

For routing purposes, the NCS needs to receive topology and network state information. This includes information about the location and utilization of potential processing nodes. Link-state protocols such as OSPF [7] and PNNI [1] provide information about the network topology and, in the case of PNNI, available bandwidth of links. These can be extended to include available processing resources at active nodes and attribute information. For reasons of scalability, this information must be aggregated, like in PNNI.

As a result of state aggregration and potentially stale information, the determined route of a session and the locations of the code modules might not be optimal or even feasible. To deal with this situation, the signaling subsystem needs to support crankback mechanisms to find alternate routes.

The module property and module code databases can be organized in a hierarchy with several levels of caching, like the DNS system, to ensure scalability.

V. Related Work

In the context of active networks, resource discovery and resource reservation are crucial factors of network programmability. Darwin [4] proposes an integrated resource management scheme that maps application constraints onto the network resources. The application provides resource requests in form of a virtual mesh, which is an annotated graph structure that specifies desired high-level services as nodes in the graph. A resource broker (Xena) then translates the virtual mesh onto network resources by expressing it as a boolean optimization problem, which is generally NP-hard, thus this approach is only appropriate for small to medium sized networks. Resource discovery mechanisms have not been implemented in the Darwin system.

Chae et al. [3] propose a mechanism that allows discovery of topological properties related to network services and resource states. Internal properties of the network are collected using network queries, which are evaluated in a distributed fashion and the aggregate result is sent back to the source. This might be a useful scheme for discovering attributes of nodes and links in our architecture. The amount of information that has to be dissem-

inated over the network for attributes might be large. The queries can be restricted to the attributes that are of interest for a particular session setup.

The PNNI protocol [1] is a distributed resource allocation system for ATM networks. PNNI consists of two protocols, a hierarchical link-state protocol that distributes information about network resources, and a signaling protocol that uses this information to reserve network resources along a precomputed path. In PNNI, the source initiates a session setup by sending a request to the switch connected to the source. The switch uses information about the network topology and resource availability (which is gathered by the link-state protocol) and computes a path to the destination. The switch then passes the selected route to switches along the path and each switch then tries to allocate local resources. If an attempt to make a reservation fails, a crankback process is initiated and an alternate path to the destination is computed. To make this approach scalable to very large networks, switches have only complete knowledge about their local network but aggregated information of distant networks.

VI. Conclusions

Network applications can significantly benefit from computational capabilities within the network because they are able to deploy customized packet processing services rather than depend solely on end-to-end mechanisms. Such applications need an abstract way of specifying session requirements that can be translated onto network resources.

In this paper, we propose *active network pipes* as an abstraction used by applications to specify session requirements in a high-level and expressive way. An active pipe is a sequence of functions that are executed on the data stream which is routed through the network. The locations of these functions within the network are defined through installation conditions. We present an algorithm that maps high-level application requirements onto physical network resources while guaranteeing lowest network costs. Our scheme transforms an active pipe specification into a layered graph whose solution represents the locations for modules and optimal path through the network transiting all modules. We also identify components that are needed for an implementation and propose a network software architecture that supports active pipes.

We believe that providing one common programming abstraction to various heterogeneous active network architectures is important for making active applications easily programmable and widely usable. Active network pipes represent a significant step towards the solution of this crucial problem.

References

[1] ATM Forum Technical Committee, *Private Network-Network Interface Specification Version 1.0*, March 1996.

[2] Andrew T. Campbell, Herman G. De Meer, Michael E. Kounavis, Kazuho Miki, John B. Vicente, and Daniel Villele, "A survey of programmable networks," *Computer Communication Review*, vol. 29, no. 2, pp. 7-23, Apr. 1999.

[3] Youngsu Chae, Shasi Merugu, Ellen Zegura, Samrat Bhattacharjee, "Exposing the Network: Support for Topology Sensitive Applications", *Proceedings of IEEE Openarch 2000*, March 2000.

[4] Prashant Chandra, Allan Fisher, Corey Kosak, T.S. Eugene Ng, Peter Steenkiste, Eduardo Takahashi, Hui Zhang, "Darwin: Resource Management for Value-Added Customizable Network Service", *Sixth IEEE International Conference on Network Protocols*, October 1998.

[5] Prashant Chandra, Allan Fisher, Peter Steenkiste, "Beagle: A Resource Allocation Protocol for an Advanced Services Internet", *Technical Report CMU-CS-98-150*, August 1998.

[6] Sumi Choi, Jonathan Turner, Tilman Wolf, "Configuring Sessions in Programmable Networks", *submitted to Infocom 2001.*

[7] J. Moy, *RFC 2328 OSPF Version 2*, IETF Network Working Group, April 1998.

[8] David Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, Gary J. Minden, "A Survey of Active Network Research", *IEEE Communications Magazine*, Vol. 35, no. 1, pp. 80-86, January 1997.

[9] Roch A. Guérin, "QoS Routing in Networks with Inaccurate Information: Theory and Algorithms," *IEEE Transactions on Networking*, Vol. 7, No. 3, June 1999.

[10] Ralph Keller, Sumi Choi, Dan Decasper, Marcel Dasen, George Fankhauser, Bernhard Plattner, "An Active Router Architecture for Multicast Video Distribution," *Proceedings of Infocom 2000*, Tel Aviv, March 2000.

[11] Konstantinos Psounis, "Active Networks: Applications, Security, Safety, and Architectures," *IEEE Communications Surveys*, First Quarter 1999.