DISSERTATION PREPARATION APPROVAL FORM

Title of Dissertation: A FRAMEWORK FOR COMPLEX COMPOSITION OF NETCENTRIC SYSTEMS

I. To be completed by the student:

I certify that this dissertation meets the preparation guidelines as presented in the Style Guide and Instructions for Typing Theses and Dissertations

_____ _____

(Signature of Student) (Date)

II. To be completed by Chair or Dissertation Committee:

_____ This dissertation is suitable for submission to the College

_____ This dissertation should be checked for conformity with the College guidelines

_____ _____

(Signature of Dissertation Chair) (Date)

III. To be completed by the Director of the School or Chair of the Department:

I certify, to the best of my knowledge, that the required procedures has been followed and the preparation criteria have been met for this dissertation.

_____ _____

(Signature of Director/Chair) (Date)

YANG, SEUNG S., Ph.D., December, 2004 Computer Science

A FRAMEWORK FOR COMPLEX COMPOSITION OF NETCENTRIC SYSTEMS, 1999-2004 (pp.)

Director of Dissertation: Javed I. Khan

The first generation of active networking research has demonstrated very simple programmable systems, where simple code modules can be executed in network routers and endpoints. However, complex netcentric systems require much more sophisticated design and composition formalism of active network codes. This dissertation presents a netcentric active service composition formalism which particularly focuses on code usability by polymorphism and inheritance. It demonstrates a novel active programming construct called made-to-order channels, which allows a collection of active code modules composed within such a construct to be recursively used within higher order instance of similar construct- thus enabling code reusability for active systems through a socket like interface. The construct allows active services to be incrementally layered to create versatile yet custom communication transports. The power of the formalism has been demonstrated via implementing a set of novel network transport services such for concurrent communication channel, and self-organizing-video-transcoding channel, et cetera. This dissertation has developed the formalism, demonstrated an active network platform for deployment and construction of such complex channels, and has evaluated the performance of the channel systems. The result has implication into other emerging paradigms of programmable netcentric systems such as web services, grid and autonomic computing.

A FRAMEWORK FOR COMPLEX COMPOSITION OF NETCENTRIC SYSTEMS, 1999-2004

A dissertation submitted
to Kent State University in partial
fulfillment of the requirements for the
degree of Doctor of Philosophy

by

Seung S. Yang

December 2004

Dissertation written by
Seung S. Yang
B.E., Soongsil University, 1993
M.E., Soongsil University, 1995
Ph.D., Kent State University, 2004

Approved by

_____, Chair, Doctoral Dissertation Committee

_____, Members, Doctoral Dissertation Committee

_____

_____

_____

Accepted by

_____, Chair, Department of Computer Science

_____, Dean, College of Arts and Sciences

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF TABLES

Dedication

To my wife, Hyunjung, my children Jungmo and Chelsea,

my parents, Junseag Yang and Youngsoon Min.

Their efforts and sacrifices made this dissertation possible.

Acknowledgements

Completing doctorate degree is much harder than I expected at the beginning. If I were alone, I could not finish it. Most of all, I wish to express my heartfelt thanks for care and thought my advisor and mentor, Dr. Javed I. Khan. Without his guide, I could not complete this work. He allows me to do what I do with enjoyment and confidence. He is not only my advisor but my compass for my life. Dr. Kenneth Batcher and Dr. Hassan Peyravi are my models as a researcher and teacher. I have great help from them more than they can imagine. I am proud that such prominent scholars as my dissertation committee member.

My appreciation also goes to all my friends who encouraged and helped me to complete my degree. In particular Changyong Jung who was always with me when I am happy or embarrassed and need helps.

Last but not least, I want express my great appreciation to my sister, Haekyung. She helped me to come to the United States and let me start my degree in Kent State University.

CHAPTER 1

Introduction

The current data communication network has enough transmission speed to handle not only simple data but also complex data such as multi-media content. The network also has enhanced transmission reliability to support reliability-critical applications such as network attached storage (NAS) and storage area network (SAN). Many netcentric systems, for example, Content Delivery Network (CDN), Content Service Network (CSN), distributed computing, and Grid computing exploit the enhanced network speed and reliability to build a service in a network. [22][23]

The netcentric systems construct a service by connecting their distributed software components to a network. To perform their service, the netcentric systems require customized communication features such as application specific routing-path control, application categorized delay and jitter control, real-time communication speed and error detection, fault-resilient transmission, and data forwarding rate control. [34][35][36] The development of a netcentric system on top of a simple data forwarding service partially fulfills the netcentric system's requirements. [12][13][14] Therefore, a netcentric system developer builds the application specific communication features into his software components. However, it only can react on communication messages at the communication end point in a netcentric system's component, although many application specific transmission features need custom actions along the communication path in

network. Even when a developer implements custom communication features, he has limited tools such as the socket library for building the features.

An active network supports a netcentric system developer by expanding the network's classical roles from simple data forwarding to an application specific programmability in a network. [13] In the active network paradigm, the routers or switches of the network perform customized communication features such as gathering and merging data from a set of senders (Concast), content aware gateway service (CAG), and dynamic video transcoding. [12][37] An active network, on the other hand, supports very little reusability on the construction of its own network embedded components. Until now, an active network has not provided code reusability at the level that CORBA or Globus Toolkit provides but only that the degree of template or static class basis which depends on the programming language used. [22][24] Because the required components are decided at implementation time and not at deployment time, when a custom communication connection is set up, a specific network component should be loaded in a targeted active node. It cannot use similar components which support required functions. Therefore maintaining required software components as well as deploying them is another heavy task in active network architecture.

This dissertation presents a developed framework for building a complex netcentric system that supports the following features to facilitate development of a netcentric system:

- Platform and language independent reusability on network embedded components
- Simplification of building a complex communication channel by recursive channel composition
- Enhanced utilization of network embedded components by service function based deployment time component binding
- Resource adaptive custom communication channel construction over distributed computation nodes

The framework also has features such as dynamic pathway planning, autonomous custom service construction, interfacing with service subscriber application and network, embedded system monitoring and visualization support, and dynamic interpretation of service description and service component discovery.

This dissertation is organized in the following way. In section 2, related works on the netcentric system and the needs for a new approach are presented. In section 3, design issues in the netcentric system framework are explored. Service construction formalism and optimization issues are presented in section 4. The netcentric made-to-order (MTO) channel system architecture is shown in section 5. In section 6 and 7, a single MTO channel and a multi-level MTO channel construction methods and sequences are covered. Application perspective considerations are discussed in section 8. In section 9, implemented systems and their experimental results are shown. Finally features and results of the developed framework are discussed in section 10.

CHAPTER 2

Netcentric Systems

Many aspects of the programmable network have been investigated since it drew attention. The active network paradigm is one direction that opens a way to process packets in network. [15] Many works have been done and continued to support easy and robust programmability in an active network. [39]

In this section, netcentric systems will be explored, and design considerations of our active made-to-order (MTO) channel system will be presented.

2.1 Related Works

2.1.1 ANTS

ANTS by Wetherall, et al. provides a Java-based network protocol-building and deploying framework. [14] They built an architecture that supports automatic and dynamic deployment of new protocols. The architecture deploys a new protocol as it is needed, and the protocol does not have interaction with previous deployed network protocols. The ANTS protocol architecture uses capsules, code groups, and protocols. Figure 1 (a) shows a capsule composition hierarchy. A capsule is a generalized replacement for a packet. A capsule shows which forwarding routines are used to process

the capsule. A code group that is forwarded as a unit by the code distribution system is a collection of related capsule types. A protocol that is handled as a unit in an active node is a collection of related code groups.

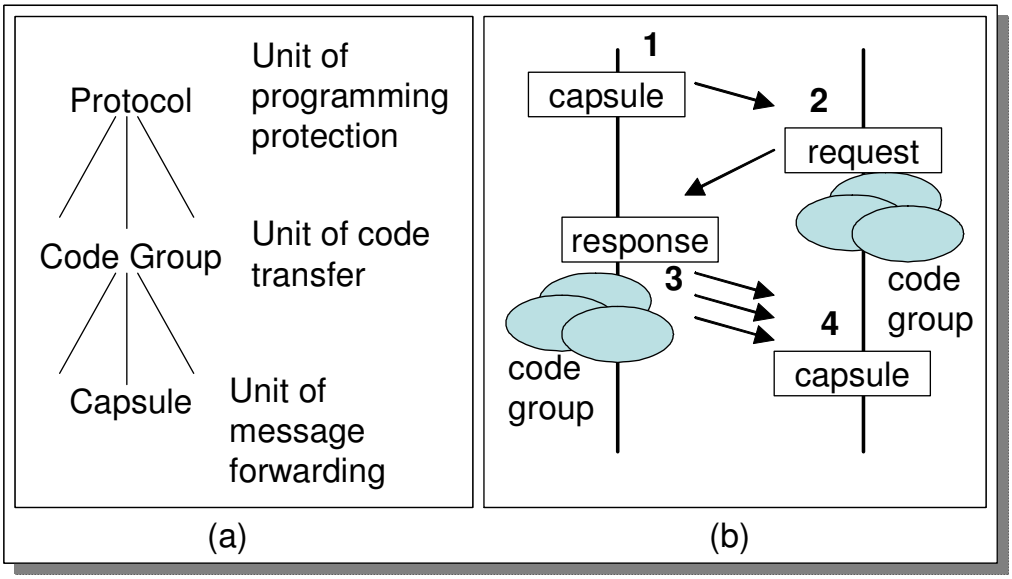

Figure 1 The Capsule Composition Hierarchy and Demand Loading of Code Groups

When forwarding routines for a capsule are not available in an active node, the active node loads the code group from its previous node and processes the capsule as shown in Figure 1 (b).

```
package apps;
import ants.*;

public class PMTUProtocol extends Protocol {
  public PMTUProtocol() throws Exception {
    startProtocolDefn();

    startGroupDefn();
    addCapsule("apps.PMTUCapsule");
    endGroupDefn();

    endProtocolDefn();
  }
}
```

Figure 2 ANTs PMTUProtocol Code

The ANTS system developed using the Java programming language. [37] The ANTS toolkit includes four base classes: Node, Protocol, Application, and Capsule. Each active node is represented by an instance of the Node class. The Channel class is used to instantiate each network interface. An application is developed by using the Application class. A new service is developed by subclassing the Capsule and Protocol classes. An ANTS developer makes his own custom protocols by using the base classes. An application developer who uses developed services writes his application programs by subclassing the provided Application class and uses the service by instantiating the developed protocol class. Figure 2 shows a sample ANTS protocol source code. The PMTUProtocol uses the Protocol class as its base class and defines its capsule group.

2.1.2 CANEs

Sanders, et al. suggest a slot processing model, CANEs, on top of Bowman Node OS. [16] A CANEs execution environment (CANEs EE) supports reasonable forwarding performance to active applications and modular service construction. It has two goals: to support the development of active applications which require reasonable forwarding performance and to provide a framework for the modular service construction. The CANEs EE has two parts, a fixed part that represents a uniform processing applied to every packet and a variable part that represents a customized functionality for the packets. The variable part, called an injected program, may be node-resident or is loaded from a remote site. The fixed part, called an underline program, declares shared variables, allocated spaces, and exported variables. The injected program imports the shared variables of its underlying programs to create bindings between the shared variables and the references. A modification to the variables is copied into local variables for use after the processing ends.

```
canes_referenced_global_per_flow (canes_packet_t *, cur_pkt);
void add_ip(void)
{
  canes_packet_t *      pk;
  struct o_if *         interface;
  ...
  pk = c_lp(cur_pkt);
  …
  channel_data_ptr = (channel_data_ptr_t)&pk->buffer[route_info_offset];

  ...
  generic_addr aa;
  boolean i = oif_find_addr_by_channel(pk->from, addrtype_ip, &aa);
  memcpy(&channel_data_ptr->hops[num_hops].ip_addr, &aa.addr.ip, aa.l);
...
  /* update the info in the headers */
  pk->size += sizeof(channel_info_t);
} /* End add_ip */

void _entry(void)
{
  cur_pkt = canes_import_global_per_flow("Cur-Pkt");
}
```

Figure 3 Add IP Injected Program

The CANEs introduces a notion of the slot where an injected program resides and runs on it. [38] A CANEs EE uses shared variables to communicate among its active programs. Shared variables are exported and imported among active programs and used as communication points. The CANEs' example systems are developed in the C programming language, even though it is not limited to a specific programming language. Underlying programs export shared variables, and injected programs import the shared variables to make connections. A developer develops his own underlying and injected programs. A developed program registers its shared variables globally to expose them and to be used by other programs. Developed programs come with CANEs user interface (CUI) messages. The CUI messages contain the path to the underlying and injected

CANEs code. CUI files for an injected program should be sent to an active node in order to run a program. The application that uses a CANEs packet uses CANEs APIs to send and receive a packet. In the packet, channel identification is included to dispatch the packet to a proper service. A sample injected program is shown in Figure 3.

2.1.3 PLAN and SNAP

PLAN and SNAP which were suggested by Hicks and Moore, et al., entail a language-based programmable network approach. [19][20] The programs are lightweight and restricted in functionality. A PLAN packet contains a field called chunk. The Chunk consists of a code written in PLAN language: an entry point which indicates the first function to execute and bindings which describe arguments for an entry function. When each active node receives a PLAN packet, a PLAN interpreter interprets the packet and runs the codes in the chunk field. SNAP is currently a developing system which uses a special purpose language and virtual machine interpreters.

A PLAN has two phases of software development: one for PLAN services and the other for PLAN packets. A PLAN service is a PLAN program which is called by codes in a PLAN packet. A PLAN service is installed manually or dynamically in an active node. The Java or the OCaml programming language is used for developing a PLAN service.

```
fun ping (source:host, destination:host, outgoing:bool) : unit =
  if outgoing and (thisHost () = destination) then
    (OnRemote (ping (destination, source, false),
            source, getRB (), defaultRoute)
    )
  else
    if not outgoing and (thisHost () = destination) then
      print ("Success")
    else OnRemote (ping (source, destination, outgoing),
                destination, getRB (), defaultRoute)
```

Figure 4 A PLAN Program: Ping

A PLAN program is written in the PLAN programming language developed for the PLAN packet. It consists of two components: a code and an invocation. The code consists of a series of definitions. The definitions bind names to abstractions, which take the form of functions, values, and exceptions. The invocation consists of the function calls to be evaluated at the next evaluation point. A PLAN developer uses the PLAN programming language for PLAN packets. Figure 4 shows a ping program written in the PLAN language.

2.1.4 Netscript

Silva, et al. introduce Netscript, which is for developing active service protocols by using a high-level programming language and environment. [21] It provides means to build a service from primitive operators by layering active elements above active or non-active components to use their services. It supports dynamic changes in its dataflow; a

box, which represents Netscript computation, can be added, removed, connected or disconnected at runtime.

```
box template RTMP
{
  box import “http://cs.columbia.edu/ns/boxlib/RTMPControl.nbt” RTMPControl
  box import “http://cs.columbia.edu/ns/boxlib/IP_RTMPMuxDemux.nbt” IP_RTMPMuxDemux
  box import “http://ietf.org/ns/boxlib/IP.nbt” IP;

  connect
  {
    IP.rcvUp-> IP_RTMPMuxDemux.rcvDown,
    IP_RTMPMuxDemux.sndDown -> IP.sndUp,
    IP_RTMPMuxDemux.rcvUp ->RTMPControl.rcvDown,
    RTMPControl.sndDown -> IP_RTMPMuxDemux.sndUp,
  }
}
```

Figure 5 A Netscript Program

A Netscript service consists of boxes and connections among the boxes. The boxes are computational parts, and connections describe the connections among the boxes. A Netscript service developer describes boxes and their connections in Netscript language. A box imports other boxes. It also includes a connection section which describes the connections among the imported boxes. Figure 5 shows a sample Netscript program. In the sample program, a box is implemented in a Java bean which includes SndUp, SndDown, RcvUp, and RcvDown interfaces to communicate with other boxes. The box can use templates to extend an existing protocol.

2.1.5 Distributed and Grid Computing

The active network's success seems to still be limited to lower network functionality with limited state programmability. It has limited support for the reusability on pre-developed component such as language based or static network component reusability. Distributed and Grid computing, on the other hand, explored high level programming reusability and enabled better software component recycling. Their service construction methods are enhanced dramatically. A software developer uses well designed infrastructures such as CORBA, and Globus Toolkit to build his services on Distributed and Grid computing. [22][24] Open Grid Services Architecture enables the integration of services and resources among distributed and heterogeneous organizations. The Globus Toolkit addresses issues of security, information discovery, resource management, and portability. [23] With the infrastructures, a service designer can concentrates more on his work.

However, when the service designer connects his service components through network, he still builds his network connection on a legacy network; he takes advantage on elegant software object reusability but has lack of application oriented communication control supports from a network. He develops required features at the communication end points in his components. Because of the consolidation of custom communication controls into a software object, the developed communication features are reused only thorough a static method such as a library. The location and reusability limitation causes

developer's enormous efforts to hold network connection controls even in Distributed and Grid computing.

2.2 Proposed System Approach

The current ongoing netcentric systems have entailed an enormous amount of work to set up a programmable network architecture. Their architectures support a dynamic load-ability of software components on an active node and a modularity of plug-in softwares.

Table 1 Netcentric Systems' Features

	ANTS	CANEs	PLAN	NetScript
Code Location	Outside	Outside	Inside	Outside/Inside
Composition Language	Java	LIANE	PLAN	NetScript
Service Construction Language	Java	C	Java/OCaml	Java
Code Loading Location	Previous node	Code Server	In Packet	In Packet
Code Installation Time	Before Capsule Processing	Before a Service Start	With Packet	With Packet
Required Code Resolving Time	Service Development Time	Service Development Time	Service Development Time	Service Development Time
Resource adaptation	Single node only	Single node only	Single node only	Single node only
Event propagates	No	No	No	No

Table 1 shows netcentric systems' features. The netcentric systems have programmability on network connection, a dynamic code loading mechanism,

interference free a new protocol support from existing protocols. However, their architectures only have language-based reusability, or limited network component reusability by binding required code at development time while distributed and Grid computing support very high level reusability on their software modules but not in their communication objects. Even when the netcentric systems deploy their network embedded components on active nodes, they are deficient in systematical consideration of available resources in overall active nodes in a network.

Information division and binding are required when a system is dynamically configuring a service with deliberation of network resources. The presented netcentric systems have lack of information division and binding features. They fixed service components for processing packet at a programming time when they have limited knowledge of actual resources and user's requirements at their running time. This causes a less adaptive deployment planning of a service system and reduces dynamicity of system configuration.

Here an Active Made-to-Order Channel System (AMCS) that supports a dynamic MTO channel construction and management framework with MTO channel reusability is presented. A MTO channel is a transmission object in between two or more communication entities. Examples of a MTO channel and its usages are explored. The AMCS' recursive channel construction method leverages reusability of MTO channels when the system build a complex MTO channel. The AMCS has the following capabilities:

- Dynamic MTO channel components loading/unloading
- Reusability on MTO channels
- Dynamic binding of MTO channel and sub-MTO channel during run time
- Adaptation to run time system
- Embedded system status monitoring support

The AMCS system loads MTO channel components when a service is initiated. The binding of a MTO channel name to an actual MTO channel is performed at a service construction time not a service development time. This increases independence and flexibility of the service construction. The separation also enhances maintenance on service components that is not supported by static components binding protocols. The AMCS system augments the reusability of a developed MTO channel by using it as a sub-component of another MTO channel service. The AMCS system can constructs a service over distributed nodes. It is different from most other systems which construct a service on a node not over distributed nodes. Using the service construction, the proposed system builds a service which can not be performed with resources on single node.

In the following sections, netcentric system design considerations, the MTO channel system architecture, the MTO channel construction formalisms, and the MTO channel construction optimizations will be presented.

CHAPTER 3

Design Issues in Complex Netcentric Systems

A dynamic composition of complex netcentric systems is not a simple task. To consolidate the framework system, from the software engineering issues to information division and sharing were visited.

3.1 System Design Considerations

Because most previous approaches for building a complex networked system have eventually focused on building a more complex construction process using an elementary standard transport library for their connectivity, a *software engineering approach* for building a transportation channel is entailed as an important design consideration. Conceptually an active network provides a platform for building a complex communication system. The active network, however, lacks a polymorphic framework with inheritance property which includes both a process and an interaction construct. The developed system constructs a complex transportation channel by reusing other channels systematically. The channel construction should use uniform interfaces from a simple channel construction to a complex channel composition.

During a *channel set up planning phase*, the designed netcentric system provides a deployment time mapping mechanism that entails application knowledge into a channel

construction. It is a critical requirement to get an appropriate specification language for mapping application needs into a MTO channel and network resource requirements. The description of the application needs to be interpreted and used to find a specific service MTO channel which supports the requirements. Also, the service description is bonded with a MTO channel's resource requirement description, and the system/network resource requirement description will be produced.

The netcentric system supports *accessing network states*. In classical design, each layer in the network software stack has been strongly isolated from its upper layer. This model simplified the development of the first generation applications. The flip side of the design choice is that now network states are very difficult to access. Therefore, the upper layer cannot take advantage of reacting to the dynamic network status changes. The designed netcentric system provides uniform interfaces to access dynamic network status information to leverage its adaptation on system/network status changes.

For building an efficient network MTO channel layer for active application, *system level re-provisioning* will be required so that embedded service components can be mutually secured and trusted, and yet will allow open standard-based access to network local states. To accommodate adaptation for network status, dynamic relocation is needed. In service reconfiguration, additional constraints are placed by the requirements of minimum interruption of the ongoing service sessions. The service relocation must reduce the impact on overall processing and communication delay.

Another critical component that arises in dynamic service composition is the creation of a powerful *middle-layer abstraction*. While the dynamic service itself can be a

complex system, the subscription and the use of the composed service should be easy and intuitive. This required provisioning is on a middle-layer abstraction, where third-party developers can develop the components required for dynamic service, while the end-point users and applications can take advantage of these services with an easy to use interface.

A *MTO channel* is a dynamically constructed transportation service system. A MTO channel represents not only a connection object among service points, but also a service system which processes information while it is in transit.

The designed MTO channel system runs on top of an active network system, even though the system's basic requirements are i) reliable control message transfer among participated nodes, ii) load and run software components in a network node, iii) reliable component delivery to a network node.

3.2 Information Division and Sharing

Multiple participants are involved in a MTO channel construction. Fractional information is supplied by each participant. A channel is constructed by dynamically binding the information during set up and run time. In this section, three participated entities are identified and examined for a MTO channel construction.

3.2.1 Information Division

The developed system has three different information providers for the MTO channel construction. The identified providers are a MTO channel developer who develops a MTO channel and its components, a user or application that uses the constructed MTO channel, and a network which supplies information for the running system/network environment. The three information provider model helps in the task separation in the development processing and is the key to any possibility of complex application engineering. First, the fragmentation will be looked, than how the system glues them together will be explained.

The MTO channel designer knows very well about the operation mode, the control flow and the detail architecture of a MTO channel. He also knows the situational constraints of the MTO channel components. However, the MTO channel designer does not have the knowledge of the exact network and system map on which the MTO channel will run. The MTO channel designer provides the MTO channel components that are required to establish a service. He also provides canonical maps that have placement information of the MTO channel components and connection information among them. In addition, he also supplies logical resource requirements of the MTO channel.

The user/subscriber application is eager to remain completely ignorant about the MTO channel architecture, components, and its connections; it is mostly interested in the service that the MTO channel provides. The subscriber, on the other hand, is the first

member to know the location of the end-points and the most knowledgeable participant among the three involved entities about the characteristics of traffic that will flow through the MTO channel. The subscriber is expected to have a fair idea about the nature of the service it will receive from the MTO channel, though it may not know the exact metric that the MTO channel designer used to quantify it. It also reserves its right to know how much it is actually getting. Consequently, it supplies service-end-points and quality of service/traffic information. However, as a user of a channel it must know about the semantics of the service. This includes the semantics of events available from a channel as well as role of some of the well known parts of the channel. Typically various component programs of a channel play distinct logical role in the realization of the transport service of the channel. A channel user is expected to know these roles though not the internal functions of the components.

The system and network environment information comes from a network through node OSes. A node OS does not have any prior knowledge about a MTO channel architecture, its components capacity, or required network capacity. It also does not know in advance about the traffic characteristics or sizes. This is, nevertheless, the most knowledgeable entity among the three about the underlying network infrastructure, i.e., the actual network topology, and the quantity and capacities of its various elements. Therefore, it provides a base topology graph and the quality-of-network information.

3.2.2 Information Binding

The information dependency is shown in Figure 6.

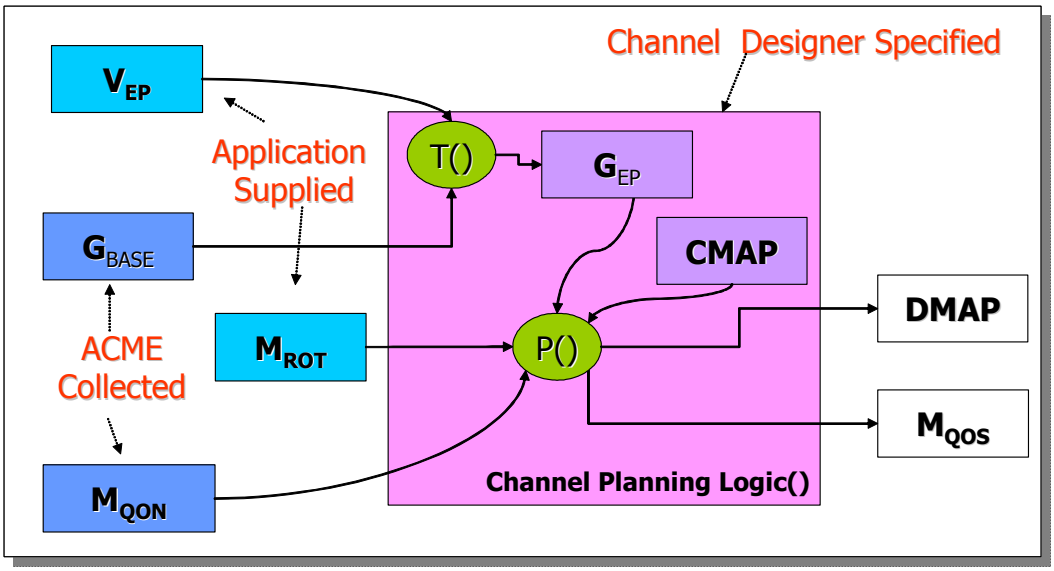

Figure 6 The Tripartite Information Dependency among the Application Program, MTO Channel, and the Network OS

The user application supplies an end-point vector (V_{EP}) and a quality-of-traffic metric (M_{ROT}). A network operating system supplies a run-time base topology graph (G_{BASE}) and a quality-of-network metric (M_{QON}). A MTO channel designer supplies a component map (CMAP), a transformation function T(), and channel planning function P(). The transformation function T() is used for generating an channel's own end-point topology graph. The end-point topology graph (G_{EP}) is computed based on a given end-point vector (V_{EP}) and a base topology graph (G_{BASE}).

The dynamic component placement map (DMAP) is calculated using the end-point topology graph (G_{EP}), the component map (CMAP), the quality-of-traffic metric (M_{ROT}), and the quality-of-network metric (M_{QON}).

Finally, the quality-of-service metric (M_{QOS}) is derived from the dynamic component placement map (DMAP), the quality-of-network metric (M_{QON}), and the quality-of-traffic metric (M_{ROT}).

A legacy TCP connection also needs information from an application, a network, and a software developer to set up its connection. A TCP connection has two pairs of IP addresses and port numbers as V_{EP}: one for source and the other for destination. The quality-of-traffic metric (M_{ROT}) in the TCP is simply a connectivity from the source to the destination. This meaning is embedded when an application requests a TCP connection. A network supports base topology graphs, G_{BASE} and $M_{QON,}$ even though those are primitive information, i.e., a partial network graph for a path of a given packet for G_{BASE} and an availability of a path from the source to the destination for M_{QON}. All the required software and functions are pre-installed in each node that participates in the communication. The component map (CMAP) are not needed for a TCP connection, and the deployment map (DMAP) is not generated for the same reason; all the software is pre-installed. A TCP connection generates G_{EP} for each packet. The G_{EP} is determined at a given time for a packet and is not persistent during the connection's lifetime. The quality-of-service metric (M_{QOS}) are the same as the quality-of-network metric (M_{QON}): providing reach-ability.

CHAPTER 4

MTO Channel Formalism

4.1 Definition of a MTO channel

A MTO channel is a transport object in between communication entities. A MTO channel is made up of computing points of presence (POP) and these POPs are connected via links. A POP can be a simple connection point or a computing component between sub-MTO channels. A special POP called a service end point (SEP) is a POP connected to an application or another MTO channel. A link is a simple connection or another MTO channel called a sub-MTO channel connecting two POPs.

Definition 1: MTO channel

A MTO channel C is a connected graph and defined as a quintuple C = (M, C', P, E, S). A MTO channel has five types of assets. M is a set of MTO channel software objects (called components). C' is a set of sub-MTO channels which are used by this MTO channel C. P is a set of points of presence (POP) where the MTO channel computation is needed to realize the MTO channel service. E is a set of event objects that relates to C. S is a set of state variables representing the MTO channel's state.

M is a set of MTO channel component that can be located in POP and performs computations on the data that flow in the MTO channel. C' is a set of sub-MTO channels which is connecting the MTO channel's components. In the MTO channel, data is delivered with or without modification of its contents based on the sub-MTO channel that it uses or on the channel components. P is a set of points of presence (POP) where the computations on the MTO channel happen. E is a set of events that is generated by the MTO channel, and which carries extra information for MTO channel system management, status reports, commands for software components, and so on. S is a set of state variables representing a state of the MTO channel.

Definition 2: Component Set

A component set of MTO channel C, M, is a set of the all MTO channel's components, $M = m_1 \cup m_2 \cup \ldots \cup m_N$. m_N *is the N-th component in channel C. A component is a software program in the MTO channel that has its own computation and/or transmission algorithms to fulfill the tasks of the MTO channel C.*

A component set of MTO channel C, M, is a set of software components in the MTO channel. The components perform their computation and/or transmission algorithms on data to realize the tasks of the MTO channel C. A MTO channel should have at least one component. A MTO channel deploys the channel components in the network and the data that is transmitted is processed and delivered by the components. All components of a MTO channel C may not be required at run time. The actual used MTO channel

components are determined dynamically at run time based on the platform of the MTO channel runs and user's requirements. Because of the dynamicity of channel construction, whole MTO channel components should be available when the MTO channel is requested.

The information provided by a channel developer comes with the channel components. The information may be delivered separately but should be provided with the channel component by the channel designer.

When an application requests a MTO channel, it provides roles of each POP in application provided information. Each role should have at least one component. Multiple components may be allocated for a single role.

Definition 3: Sub-MTO Channel Set

A sub-MTO channel set, C', is a set of MTO channels used in a MTO channel C. C' = $\{C_{s1}, C_{s2}, C_{s3}, \ldots, C_{sN} \mid$ *where* $C_{sx} \in$ *MTO channel and N is number of sub-MTO channels}$

Each sub-MTO channel in a sub-MTO channel set C' is an independent MTO channel which can be used as a stand alone. The sub-MTO channel also can use its sub-MTO channels to perform its tasks.

Definition 4: Point of Presence (POP) Set

A point of presence (POP) set of MTO channel C, P, is a set of union of all its POPs, $P = p_1 \cup p_2 \cup \ldots \cup p_N$. p_N *is a N-th POP in MTO channel C. A POP is location information where the MTO channel C's component(s) is resided and performs its functions on it. Each POP also includes role information of the POP. A role represents the logical task of the POP such as sender, receiver, and forwarder.*

A set of POPs in a MTO channel C has point of presence information in the MTO channel. The MTO channel C's component(s) is located in a POP and realizes its functions to perform the channel's tasks. Each POP not only has location information but also includes role information of the POP. A role identifies the logical task of the POP such as sender, receiver, and forwarder. The upper-level channel or application gives role information to its sub-channels for the sub-channel's POPs where the upper-level channel or application is connected to. The other POPs' roles are given by the channel while it plans its channel construction. The MTO channel component will be mapped in a POP and performed computation. Because of the dynamicity of a MTO channel, using POPs may change during the execution. There are two types of POPs, service end point (SEP) and intermediate computing point (ICP). A SEP is a POP which also can connect to the other MTO channel's SEP. An ICP only connects POPs which belong to the MTO channel's components including the MTO channel's sub-MTO channel.

An event is an information object which carries control, and management, as well as status information among MTO channel components. An event set of a MTO channel is represented in the following way.

Definition 5: Event Set

An event set of MTO channel C, E, is a set of events which includes all events generated by the MTO channel except the events processed by the MTO channel.

$E = E_c^g - E_h[C]$. *E is produce-able events of MTO channel C.* E_c^g *is generated events in the MTO channel C.* $E_h(C)$ *is processed events in the MTO channel C. An event is an object that has control or status information of the channel and that is not part of messages to be transferred as a content of the channel. An event has an event handler which processes the event. If no event handler is found, NOS performs a default action for the event that is indicated by the system configuration.*

The producible events of a MTO channel C are generated events in the MTO channel C except the events which are processed by the MTO channel C. An event name is created by a channel designer with properly structured method, so that the MTO channel system can uniformly identify and processes the event. An event is generated in a channel component, and the event is processed by its event hander. If the event handler is found in the same MTO channel, then the event is consumed by the event hander. If an event hander is not found in the same MTO channel, the event is propagated to its upper-level channel. The event that is propagated to the upper-level channel is producible events of

the MTO channel C. The producible events delivered their upper-level channel or processed by a node OS when no event handler is found for those events.

Definition 6: State Set

A state set of MTO channel C, S, is a set of all its states, $S = s_1 \cup s_2 \cup \ldots \cup s_N$. *N is the index of a state in MTO channel C.* s_N *is the N-th state of the MTO channel C. A state is an object that represents a condition or stage of the MTO channel C. The states and the set of states are provided by the MTO channel C's designer.*

Each MTO channel has its own channel states. A state is an object that represents a condition or stage of the MTO channel C that is defined by the channel designer. A set of MTO channel's states is also defined by the channel designer and provided with the channel component. The representation of state should be properly structured to be interpreted uniformly in the entire system. A state of a channel is independent from its sub-MTO channel's state. However, some common states such as sts_full_setup and sts_terminate are included in all channels. The MTO channel designer also provides interfaces to access the state of a MTO channel, so that an application or upper-level channel can query the state of a channel for sophisticated control and management on its traffic.

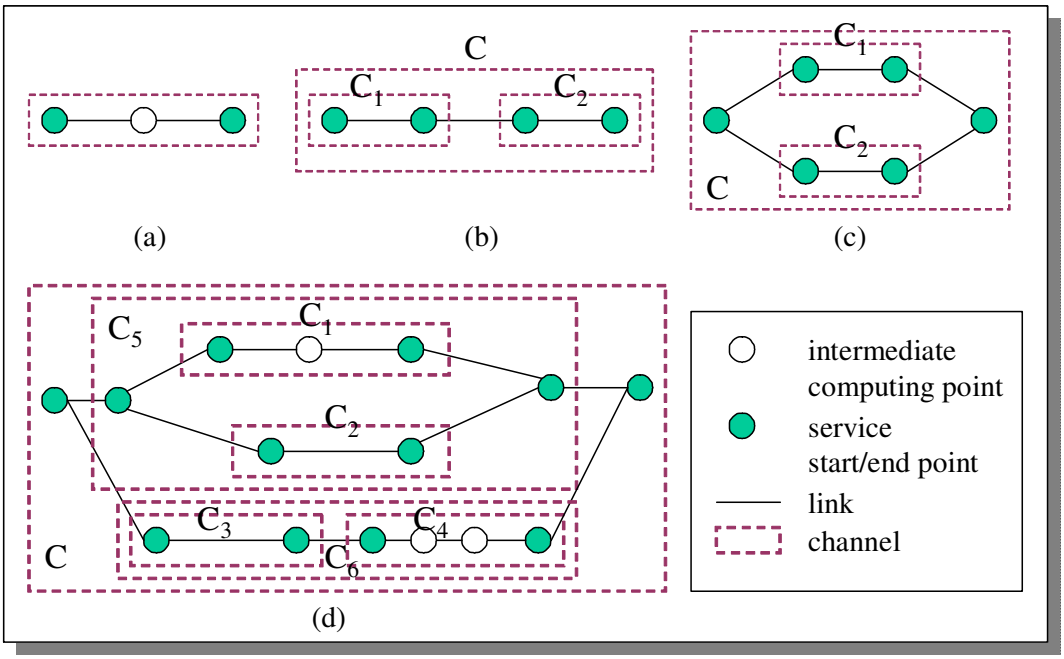

Figure 7 MTO Channel Examples

Some MTO channel examples are shown in Figure 7. Figure 7(a) is a simple MTO channel. It has two service end points, and a point which resides in between the SEPs. Figure 7 (b) is a MTO channel which combines two MTO channels sequentially. In other words, the MTO channel C uses two sub-MTO channels, C1, C2, and connects them in sequence. So the contents of the communication should pass through the MTO channels. Figure 7 (c) is a MTO channel combined with two MTO channels in parallel. The data sending from one of the SEPs of the MTO channel C may use one of the sub-MTO channels or both depending on the MTO channel C's property. Figure 7 (d) is a complex MTO channel which uses sub-MTO channels and combines them sequentially and/or in parallel.

In this paper the hierarchy of a MTO channel is represented by using sub-MTO channels enclosed by parentheses. For example, the Figure 7 (e) can be represented $C(C_5(C_1,C_2),C_6(C_3,C_4))$.

4.2 MTO Channel Component Types

4.2.1 Type by purpose

Components of a MTO channel can be classified by their purpose in the MTO channel. There are three types of computing components: manager, scout and service.

A *manager* component is a representative component in a MTO channel. It is an interface point to its upper layer or an application. A manager component sets up its other components as required and terminates the MTO channel when needed. It coordinates with its sub-MTO channels to provide a custom service. Also, it handles events and errors.

A *scout* component is a component which may be deployed before a MTO channel is fully set up. A scout component is used for gathering information in a network node for giving information which is required for a MTO channel, i.e. a computational capability in a node, or available network bandwidth.

A *service* component handles the actual data transfer. It serves actual data transmission and supports features of the MTO channel. It also reports exceptions to the MTO channel manager and produces events, but actual handling of an exception or an event is decided by the MTO channel manager. An IP forwarder is an example of simple services. A MTO channel usually has at least two service components – sending and

receiving endpoints. Each of the service components typically plays a distinct logical role in the realization of the transport service of the channel. A channel user is expected to know these roles though not the internal functions of the components. An IP forwarder is an example of simple services. A MTO channel usually has at least two service components (or roles) – sending and receiving endpoints.

4.2.2 Type by location

Components of a MTO channel are also classified by their location. There are two types of locations: service end point (SEP) and intermediate computing point (ICP).

A service end point (SEP) component is located at the end of a service; it has communication facility with the MTO channel user/application. A SEP component needs to monitor its connection to the nearest MTO channel component for connection event handling.

An intermediate computing point (ICP) component is located at the intermediate nodes between SEPs. It forwards contents of the MTO channel communication and actively applies custom processes to the contents.

Usually a MTO channel manager and application interface components are located in SEP. Many service components are generally located in the ICP with coordination of a MTO channel manager.

4.3 Recursive MTO Channel Construction

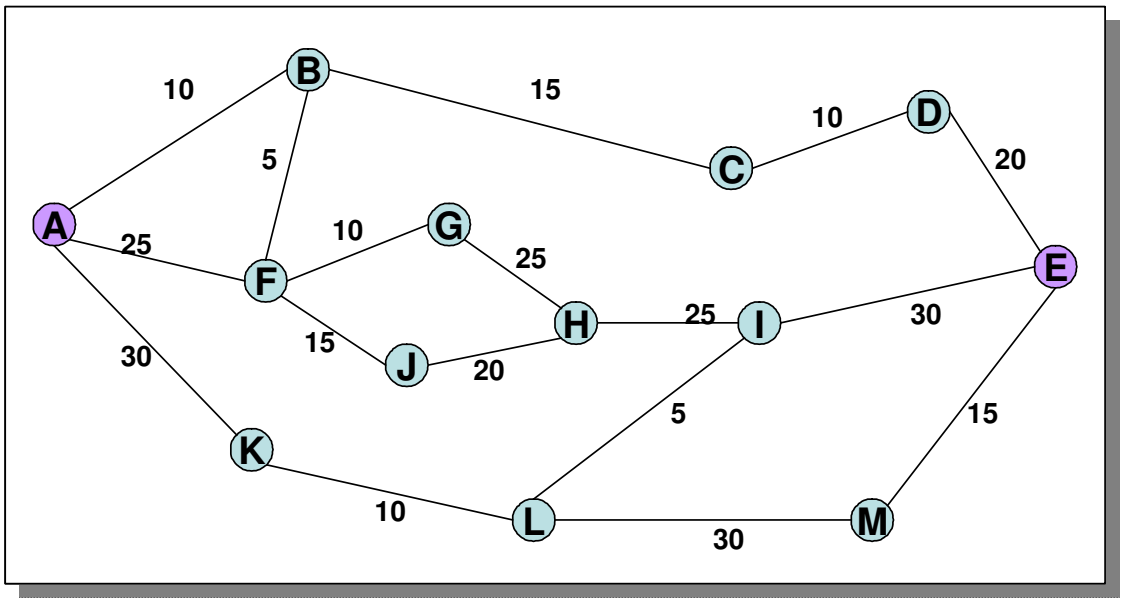

Figure 8 Example Network Diagram: Network Diagram in between A and E Nodes

An example network diagram is shown in Figure 8. Numbers above a link represent bandwidth of the link in Mbps. A user needs a MTO channel from A to E with 20 Mbps bandwidth. A general active channel constructor extension layer parses the user's requirement and will map each requirement to a MTO channel service, i.e. 20Mbps – A graph MTO channel service. Note in this example conventional TCP or UDP transport will not be able to sustain the required capacity given that classical IP only uses a single default path. However, it can be shown that by using graph wide concurrent communication and custom transport the bandwidth can be easily satisfied. This example is now used to illustrate the concept of recursive construction.

The demonstrated solution uses three MTO channel: Graph Channel (GC), Altered Routing Channel (ARC), and Concurrent Channel (CC). The GC can use and coordinate ARC and CC to set up a required bandwidth channel. The ARC uses TCP between segments of given path to support the bandwidth. If a segment in the path cannot be connected with a TCP connection to support the bandwidth, it uses GC to connect the segment. The CC sets up a connection using multiple ARC channel to support the required bandwidth. The definitions and descriptions of the three MTO channels are now given:

Graph Channel: *GC = ({gc_mgr, gc_chk}, {"Altered Routing Channel", "Concurrent Channel"}, {"source", "destination"}, {evt_connected, evt_disconnected, evt_resource_changed, evt_sub_channel_error}, {sts_initial, sts_sub_setup, sts_full_setup, sts_suspend, sts_terminate}).*

The gc_mgr is the graph MTO channel's manager component. The channel has gc_chk scout component and uses an altered routing MTO channel or a concurrent MTO channel as its sub-channel. It has two POPs: source and destination. The channel has four events: evt_connected, evt_disconnected, evt_resource_changed, and evt_sub_channel_error as well as five channel statuses.

Altered Routing Channel: *ARC = ({arc_mgr, arc_chk, arc_src, arc_ctr, arc_dst}, {"Generic Channel"}, {"source", "intermediate", "destination"}, {evt_connected, evt_disconnected, evt_segment_failure, evt_resource_changed, evt_sub_channel_error}, {sts_initial, sts_sub_setup, sts_full_setup, sts_suspend, sts_terminate, sts_num_hops}).*

An instance of “Altered Routing Channel,” ARC, MTO channel sets up a connection with a given path that may not the first preferred path from the source to the destination. The ARC MTO channel has five components: arc_mgr as a manager component, arc_src, arc_ctr, and arc_dst as service components, and arc_chk as a scout component. The ARC MTO channel may use a generic MTO channel, GC, as its sub-channel.

Concurrent Channel: *CC = ({cc_mgr, cc_chk, cc_src, cc_dst}, {“Altered Routing Channel”}, {“source”, “destination”}, {evt_connected, evt_disconnected, evt_path_failure, evt_resource_changed, evt_sub_channel_error}, {sts_initial, sts_sub_setup, sts_full_setup, sts_suspend, sts_terminate, sts_num_paths}).*

A concurrent channel, CC, sets up parallel connection(s) that does not have overlapped paths from its source to its destination. It has a manager component (cc_mgr), a scout component (cc_chk), a service component in its source location (cc_src), and a service component in its destination location (cc_dst). The cc_src and the cc_dst have multiplexing and demultiplexing functions on them to distribute and concatenate communication messages in the channel. It uses altered routing channels to set up required parallel paths from the source to the destination.

Now a step by step explanation is given how a recursive process can use combination of the proposed three MTO channels to create a complex transport under the constraints of information division explained in section.

The preliminary step to use a channel is that a channel designer provides a channel definition, such as GC = ({gc_mgr, gc_chk}, {“Altered Routing Channel”, “Concurrent Channel”}, {“source”, “destination”}, {evt_connected, evt_disconnected,

evt_resource_changed, evt_sub_channel_error}, {sts_initial, sts_sub_setup, sts_full_setup, sts_suspend, sts_terminate}) and channel components, such as gc_mgr, gc_chk, and a component map such as "c-order:gc_mgr, i-order: (gc_chk) gc_mgr." The component map is supplement information that provides channel component connection and invocation order information. The designer can supply the component map as imbedded in the channel manager or supplied separately.

Step 1: When an application requires "a connection from node A to node E with 20Mbps bandwidth," the application invokes GSocket function in the GRACE library to set up a channel. The GRACE library interprets the semantic, "bandwidth channel," to graph MTO channel (GC) and the constructs an application information metric for GC(A,E). The GRACE invokes the GC MTO channel and passes the application information metric to the channel. Note that the application information metric is a special data structure which contains the information elements identified in Table 2. As stated the invoking entity is expected to specify the end point locations, and perhaps few additional well known role placement points of the requested sub-channel.

The application information metric has V_{EP} and M_{ROT}. V_{EP} has role and location information for the channel. The location information has physical address of the channel point-of-presence such as IP(A) and IP(E) and logical address of the POP such as Label A and Label E. The V_{EP} also have a role of the POP. The role describes the task of the POP such as S for sender, R for receiver, and F for forwarder. At least a channel component is assigned to a role. For example a forwarding channel component is assigned to a forwarder role. Multiple components, however, can be assigned to a role to

perform the role. M_{ROT} has description of quality of traffic information. In the M_{ROT} metric the diagonal value represents throughput for in that location. The other values represent bandwidth for the traffic. The M_{ROT}, however, is not limited for the bandwidth only, but it can represent other qualities such as reliability. The M_{ROT} also can be described as a separate metric, if it is complex represents in one metric, but in this example it is presented in application information metric.

Table 2 Application Information Metric for GC(A,E)

V_{EP}	IP	IP(A)	IP(E)
	Role	S	R
	Label	A	E
M_{ROT}	A	20	20
	E	20	20

In M_{ROT}, the value 20 in index (A,A) means it needs 20Mbps throughput in the node A that is same in index (E,E), and the value 20 in index (A,E) presents that it needs 20Mbps bandwidth. The GRACE library creates the application information metric, invokes GC(A,E), and passes the metric to GC(A,E).

Step 2: With the application information metric, the GC(A,E) creates channel invocation information metric, requests a network information metric to NOS, and adds the result to the metric as shown in Table 3. Note that channel invocation information metric has an expanded data structure which now contains additional information elements identified in Table 3. The V_{EP} and the M_{ROT} contain the channel application information metric data, and the G_{BASE} and the M_{QON} are added from NOS. The metric

contains both G_{BASE}, network graph information, and M_{QON}, quality of network information, in one representation by representing the connection with its bandwidth and node computation capacity in diagonal.

Table 3 Channel Invocation Information Metric of GC(A,E)

V_{EP}	IP	IP(A)				IP(E)								
	Role	S				R								
	Label	A	B	C	D	E	F	G	H	I	J	K	L	M
M_{ROT}	A	20				20								
	E	20				20								
G_{BASE} and M_{QON}	A	5	10				25					30		
	B	10	5	15			5							
	C		15	5	10									
	D			10	5	20								
	E				20	5				30				
	F	25	5				5	10			15			
	G						10	5	25					
	H							25	5	25				
	I					30			25	5			5	
	J						15				5			
	K	30										5	10	
	L									5		10	5	30
	M												30	5

Step 3: Using the channel invocation information metric, the GC(A,E) checks that any single path can connect from A to E and supports the given bandwidth. However, no single path can support that bandwidth from node A to node E. The GC(A,E) divides the path from A to E using altered routing MTO channel, ARC, and checks again. Eventually the GC(A,E) checks the possibility of ARC(A,F,E). Now the GC(A,E) has the channel invocation information metric with DMAP information as shown in Table 4. The DMAP is a planned deployment map which identifies where the channel components or sub-channels should be deployed.

Table 4 Channel Invocation Information Metric of GC(A,E)

V_{EP}	IP	IP(A)				IP(E)								
	Role	S				R								
	Label	A	B	C	D	E	F	G	H	I	J	K	L	M
M_{ROT}	A	20				20								
	E	20				20								
G_{BASE} and M_{QON}	A	5	10				25					30		
	B	10	5	15			5							
	C		15	5	10									
	D			10	5	20								
	E				20	5				30				
	F	25	5				5	10			15			
	G						10	5	25					
	H							25	5	25				
	I					30			25	5			5	
	J						15				5			
	K	30										5	10	
	L									5		10	5	30
	M												30	5
DMAP	Sub-channels	ARC(A,F,E)												

Step 4: The GC(A,E) also creates an application information metric for ARC(A,F,E) and call the ARC(A,F,E) scout component (arc_chk) for checking the availability. When the GC(A,E) creates the application information metric for ARC(A,F,E), the roles of the ARC(A,F,E) is assigned by the GC(A,E).

Table 5 Application Information Metric for ARC(A,F,E)

V_{EP}	IP	IP(A)	IP(E)	IP(F)
	Role	S	R	F
	Label	A	E	F
M_{ROT}	A	20		20
	E		20	20
	F	20	20	20

Step 5: With the application information metric from GC(A,E), the ARC(A,F,E) augments its channel invocation information metric by adding network information from the node OS. The ARC(A,F,E) requests network information from NOS separately from its upper-level channel because a channel may require different network information that was used in upper-level channel and the upper-level channel does not know what information will be used in its sub-channel. In this example, each channel retrieves its own network information metric from NOS for its channel construction.

Table 6 Channel Invocation Information Metric of ARC(A,F,E) after adding Network Information

V_{EP}	IP	IP(A)				IP(E)	IP(F)							
	Role	S				R	F							
	Label	A	B	C	D	E	F	G	H	I	J	K	L	M
M_{ROT}	A	20					20							
	E					20	20							
	F	20				20	20							
G_{BASE} and M_{QON}	A	5	10				25					30		
	B	10	5	15			5							
	C		15	5	10									
	D			10	5	20								
	E				20	5				30				
	F	25	5				5	10			15			
	G						10	5	25					
	H							25	5	25				
	I					30			25	5			5	
	J						15				5			
	K	30										5	10	
	L									5		10	5	30
	M												30	5

Step 6: The ARC(A,F,E) confirmed that the connection from A to E is okay by TCP but could not decide the connection from F to E. Table 7 shows the channel information metric after it has plan for its channel module and checking the connection from F to E with GC(F,E). M_{QOS} is a quality-of-service metric that represents what service quality will be given from the channel. In this M_{QOS}, the diagonal denotes throughput of the node, and the other values represents how much bandwidth will be given in the connection. Note that in the M_{QOS} some bandwidth numbers in M_{QOS} are 25Mbps instead of 20Mbps because the channel can support up to 25Mbps in that segment.

Table 7 Channel Invocation Information Metric of ARC(A,F,E) before Invoking GC(F,E)

V_{EP}	IP	IP(A)				IP(E)	IP(F)							
	Role	S				R	F							
	Label	A	B	C	D	E	F	G	H	I	J	K	L	M
M_{ROT}	A	20					20							
	E					20	20							
	F	20				20	20							
G_{BASE} and M_{QON}	A	5	10				25					30		
	B	10	5	15			5							
	C		15	5	10									
	D			10	5	20								
	E				20	5				30				
	F	25	5				5	10			15			
	G						10	5	25					
	H							25	5	25				
	I					30			25	5			5	
	J						15				5			
	K	30										5	10	
	L									5		10	5	30
	M												30	5
DMAP	Components	src				dst	ctr							
	Sub-channel						GC(F,E)							
M_{QOS}	Computing	1				1	1							
	A	25					25							
	E													
	F	25					25							

Step 7: The ARC(A,F,E) creates an application information metric for GC(F,E) and passes the metric to it.

Table 8 Application Information Metric for GC(F,E)

V_{EP}	IP	IP(E)	IP(F)
	Role	R	S
	Label	E	F
M_{ROT}	E	20	20
	F	20	20

Even though the ARC(A,F,E) can support more bandwidth than the channel required, the bandwidth requirement for its sub-channel still remained same as its bandwidth requirement from its upper-level channel.

Step 8: The GC(F,E) is invoked from ARC(A,F,E) with the application information metric from ARC(A,F,E) and adds network information to its channel invocation information metric shown in Table 9.

Table 9 Channel Invocation Information Metric of GC(F,E) after Adding Network Information

V_{EP}	IP	IP(E)	IP(F)				
	Role	R	S				
	Label	E	F	G	H	I	J
M_{ROT}	E	20	20				
	F	20	20				
G_{BASE} and M_{QON}	E	5				30	
	F		5	10			15
	G		10	5	25		
	H				5	25	20
	I	30		25	25	5	
	J		15		20		5

Step 9: The GC(F,E) checks any single path from F to E is available for the requirement, but it is not available. The GC(F,E) divides the path and checks feasibility using ARC(F,H,E).

Table 10 Channel Invocation Information Metric of GC(F,E) before Invoking ARC(F,H,E)

V_{EP}	IP	IP(E)	IP(F)				
	Role	R	S				
	Label	E	F	G	H	I	J
M_{ROT}	E	20	20				
	F	20	20				
G_{BASE} and M_{QON}	E	5				30	
	F		5	10			15
	G		10	5	25		
	H				5	25	20
	I	30		25	25	5	
	J		15		20		5
DMAP	Sub-channel		ARC(F,H,E)				

Step 10: Because of the GC(F,E) does not have other channel component as a service component, it has only sub-channel section in its DMAP. The GC(F,E) creates an application information metric for ARC(F,H,E) and invokes the ARC(F,H,E).

Table 11 Application Information Metric for ARC(F,H,E)

V_{EP}	IP	IP(E)	IP(F)	IP(H)
	Role	R	S	F
	Label	E	F	H
M_{ROT}	E	20		20
	F		20	20
	H	20	20	20

Step 11: The ARC(F,H,E) uses the application information metric, creates its channel invocation information metric, and adds network information from NOS on the metric.

Table 12 Channel Invocation Information Metric of ARC(F,H,E) after Adding Network Information

V_{EP}	IP	IP(E)	IP(F)		IP(H)		
	Role	R	S		F		
	Label	E	F	G	H	I	J
M_{ROT}	E	20			20		
	F		20		20		
	H	20	20		20		
G_{BASE} and M_{QON}	E	5				30	
	F		5	10			15
	G		10	5	25		
	H				5	25	20
	I	30		25	25	5	
	J		15		20		5

Step 12: The ARC(F,H,E) confirmed that the connection from H to E is possible but could not decide the connection availability from F to H. So, the ARC(F,H,E) plans to check the availability of the segment with GC(F,H). The ARC(F,H,E) plans on its channel component deployments based on that it will use GC(F,H) for the segment from F to H. Note that the M_{QOS} has some 25Mbps instead of 20Mbps because the channel can support maximum 25Mbps in the segment of H to E. The ARC(F,H,E) creates an application information for GC(F,H).

Table 13 Channel Invocation Information Metric for ARC(F,H,E) before Invoking GC(F,H)

V_{EP}	IP	IP(E)	IP(F)		IP(H)		
	Role	R	S		F		
	Label	E	F	G	H	I	J
M_{ROT}	E	20			20		
	F		20		20		
	H	20	20		20		
G_{BASE} and M_{QON}	E	5				30	
	F		5	10			15
	G		10	5	25		
	H				5	25	20
	I	30		25	25	5	
	J		15		20		5
DMAP	Components	dst	src		ctr		
	Sub-channels		GC(F,H)				
M_{QOS}	Computing	1	1		1		
	E	25			25		
	F		20		20		
	H	25	20		20		

Table 14 Application Information Metric for GC(F,H)

V_{EP}	IP	IP(F)	IP(H)
	Role	S	R
	Label	F	H
M_{ROT}	F	20	20
	H	20	20

Step 13: The invoked GC(F,H) adds network information to its channel invocation information metric and checks any single path from F to H is available for the requirement. However, it is not available. The GC(F,H) checks any ARC is available for the requirement, but it also not available. The GC(F,H) checks CC(F,H) is available for the requirement. The GC(F,H) creates an application information metric for CC(F,H).

Table 15 Channel Invocation Information Metric of GC(F,H) before Invoking CC(F,H)

V_{EP}	IP	IP(F)		IP(H)	
	Role	S		R	
	Label	F	G	H	J
M_{ROT}	F	20		20	
	H	20		20	
G_{BASE} and M_{QON}	F	5	10		15
	G	10	5	25	
	H		25	5	25
	J	15		25	5
DMAP	Sub-channels	CC(F,H)			

Table 16 Application Information Metric for CC(F,H)

V_{EP}	IP	IP(F)	IP(H)
	Role	S	R
	Label	F	H
M_{ROT}	F	20	20
	H	20	20

Step 14: The CC(F,H) adds network information to its channel invocation information metric and checks ARC(F,G,H) and ARC(F,J,H) are possible for the requirement. The CC(F,H) creates application information metrics for ARC(F,G,H) and ARC(F,J,H).

Table 17 Channel Invocation Information Metric of CC(F,H) before Invoking ARC(F,G,H) and ARC(F,J,H)

V_{EP}	IP	IP(F)		IP(H)	
	Role	S		R	
	Label	F	G	H	J
M_{ROT}	F	20		20	
	H	20		20	
G_{BASE} and M_{QON}	F	5	10		15
	G	10	5	25	
	H		25	5	25
	J	15		25	5
DMAP	Components	src		dst	
	Sub-channel	ARC(F,G,H), ARC(F,J,H)			
M_{QOS}	Computing	1		1	
	F	20		20	
	H	20		20	

Table 18 Application Information Metric for ARC(F,G,H)

V_{EP}	IP	IP(F)	IP(G)	IP(H)
	Role	S	F	R
	Label	F	G	H
M_{ROT}	F	10	10	
	G	10	10	10
	H		10	10

Table 19 Application Information Metric for ARC(F,J,H)

V_{EP}	IP	IP(F)	IP(H)	IP(J)
	Role	S	R	F
	Label	F	H	J
M_{ROT}	F	10		10
	H		10	10
	J	10	10	10

Step 15: The ARC(F,G,H) adds network information to its channel invocation information metric and confirms that the connection for the path FGH supports the requirement. Now the ARC(F,G,H) completes its deployment planning and its creates channel information metric from the channel invocation information metric and returns its deployment information metric to CC(F,H). The channel information metric has same contents except that it has complete information on channel construction planning. Therefore the channel information metric has V_{EP}, M_{ROT}, G_{BASE} and M_{QON}, DMAP, M_{QOS}, and its sub-channels DMAP and M_{QOS}. Table 20 shows the ARC(F,G,H)'s channel information metric.

Step 16: The ARC(F,J,H) also confirms the availability and returns its deployment information metric to CC(F,H) too. The ARC(F,G,H) and ARC(F,J,H) are the terminal MTO channel. So, after they finalize its channel construction plan, it forwards its DMAP and M_{QOS} information to its upper-level channel.

Table 20 Channel Information Metric of ARC(F,G,H)

V_{EP}	IP	IP(F)	IP(G)	IP(H)
	Role	S	F	R
	Label	F	G	H
M_{ROT}	F	10	10	
	G	10	10	10
G_{BASE} and M_{QON}	H		10	10
	F	5	10	
	G	10	5	25
	H		25	5
DMAP	Components	src	ctr	dst
M_{QOS}	Computing	1	1	1
	F	10	10	
	G	10	10	10
	H		10	10

Table 21 Channel Information Metric of ARC(F,J,H)

V_{EP}	IP	IP(F)	IP(H)	IP(J)
	Role	S	R	F
	Label	F	H	J
M_{ROT}	F	10		10
	H		10	
	J	10		10
G_{BASE} and M_{QON}	F	5		15
	H		5	20
	J	15	20	5
DMAP	Components	src	dst	ctr
	Computing	1	1	1
M_{QOS}	F	15		15
	H		15	15
	J	15	15	15

Note that M_{QOS} is 15Mbps not 10Mbps because it can support up to 15Mbps.

Step 17: The CC(F,H) completes its deployment information after it receives deployment information metric from ARC(F,G,H) and ARC(F,J,H). The CC(F,H) creates its channel information metric by including the deployment information metric from its sub-channel at the end of its channel invocation information metric. The CC(F,H) now completes its channel construction plan, and it also forwarding its deployment information metric to its upper-level channel, GC(F,H).

Table 22 Channel Information Metric of CC(F,H)

V_{EP}	IP	IP(F)		IP(H)	
	Role	S		R	
	Label	F	G	H	J
M_{ROT}	F	20		20	
	H	20		20	
G_{BASE} and M_{QON}	F	5	10		15
	G	10	5	25	
	H		25	5	25
	J	15		25	5
DMAP	Components	src		dst	
	Sub-channel	ARC(F,G,H), ARC(F,J,H)			
M_{QOS}	Computing	1		1	
	F	20		20	
	H	20		20	
ARC(F,G,H) DMAP	Components	src	ctr	dst	
M_{QOS}	Computing	1	1	1	
	F	10	10		
	G	10	10	10	
	H		10	10	
ARC(F,J,H) DMAP	Components	src		dst	ctr
M_{QOS}	Computing	1		1	1
	F	15			15
	H			15	15
	J	15		15	15

Step 18: The deployment information of each channel forwarded to its upper level channel and finally the GC(A,E) has the channel information metric as shown in Table 23.

Table 23 Channel Information Metric for GC(A,E)

V_{EP}	IP	IP(A)				IP(E)								
	Role	S				R								
	Label	A	B	C	D	E	F	G	H	I	J	K	L	M
M_{ROT}	A	20				20								
	E	20				20								
G_{BASE} and M_{QON}	A	5	10				25					30		
	B	10	5	15			5							
	C		15	5	10									
	D			10	5	20								
	E				20	5				30				
	F	25	5				5	10			15			
	G						10	5	25					
	H							25	5	25				
	I					30			25	5			5	
	J						15				5			
	K	30										5	10	
	L									5		10	5	30
	M												30	5
DMAP	Sub-channels	ARC(A,F,E)												
ARC(A,F,E) DMAP	Components	src				dst	ctr							
	Sub-channel						GC(F,E)							
M_{QOS}	Computing	1				1	1							
	A	25					25							
	E													
	F	25					25							
GC(F,E) DMAP	Sub-channel						ARC(F,H,E)							
ARC(F,H,E) DMAP	Components					dst	src		ctr					
	Sub-channels						GC(F,H)							
	Computing					1	1		1					
	E					25			20					
	F						20							
	H					25	20		20					
GC(F,H) DMAP	Sub-channels						CC(F,H)							
CC(F,H) DMAP	Components						src		dst					
	Sub-channel						ARC(F,G,H), ARC(F,J,H)							
M_{QOS}	Computing						1		1					
	F						20		20					
	H						20		20					
ARC(F,G,H) DMAP	Components						src	ctr	dst					
MQOS	Computing						1	1	1					
	F						10	10						
	G						10	10	10					
	H							10	10					
ARC(F,J,H) DMAP	Components						src		dst		ctr			
M_{QOS}	Computing						1		1		1			
	F						15				15			
	H								15		15			
	J		15		15		15							

4.4 Complexity of a MTO Channel

During a MTO channel set up, specific MTO channels are discovered, and the discovered MTO channels are constructed on network nodes. The discovery of the required MTO channels is an unbounded problem and its complexity can reach from linear to a NP-complete problem. For example, a channel construction from two nodes in a network with all TCP connections is bounded in O(E) where E is number of edges in the network. However, a channel builds a Hamiltonian path that makes a path between two nodes of a network graph that visits each node exactly once is a NP-complete problem. In practical case, a channel construction is performed in network that has limited number of nodes and connections with small number of channel components and sub-channels. Therefore, the channel construction happens in reasonable amount of time. The MTO channel set up process, however, has not only a discovering (planning) process but also a construction process in network. The construction process of the required MTO channel is relatively new in network. The conventional network has all the software that is required to set up a connection. The MTO channel construction complexity is dominated by the number of MTO channel components required to deploy and execute. There are two types of construction costs to build a MTO channel; one for the form cost of the MTO channel, and the other cost for the assembly of the MTO channel. The form complexity is a channel construction complexity that reflects how much overhead will be endowed in network to form the channel. It includes overheads such as signaling

overhead to construct a channel in network and loading and activating channel component overheads. In most case the form complexity is dominated by how many channel components and sub-channels are needed to build a service. The assembly time complexity is a time complexity to construct a MTO channel service. Because independent channels can be constructed simultaneously, it is subjected to the depth of channel construction not the number of required channel. Later, a channel optimization algorithm is explored to reduce a channel formation complexity. The assembly time complexity is used for estimate channel construction time to set up a timer for error handling during the channel construction.

4.4.1 MTO Channel Form Complexity

Definition 7: MTO Channel Form Complexity

The form complexity of a MTO channel C, FormComplexity(C), is defined as FormComplexity(C) = (C_s,C_c). The C_S is a sub-MTO channel form complexity and C_c is a MTO channel component form complexity. The sub-MTO channel form complexity, C_s is defined as $C_s = \alpha \times \sum_{i}^{c'} FormComplexity(c_i)$ *, and the* C_c *is defined as* $C_c = \beta \times \sum_{i}^{N} FormComplexity(M_i)$*. The* α *is a constant factor for sub-MTO channel construction, and the* β *is a constant factor for the MTO channel component construction. C' is the MTO sub-channels, N is the number of channel components, and* M_i *is the i-th channel component in the MTO channel C.*

The form complexity is determining how much overhead is in the channel construction, it has two factors: sub-MTO channel form overhead and its own channel components form overhead. In general case the constant α for sub-MTO channel form complexity is greater than the constant β for the channel's component form complexity because sub-MTO channel form has to set up channel management information added to set up components in the sub-MTO channel. Also, in most cases the form complexity of each MTO channel's component is very close because the building procedure is almost the same except for the restrictions of components and the size of channel component to deploy.

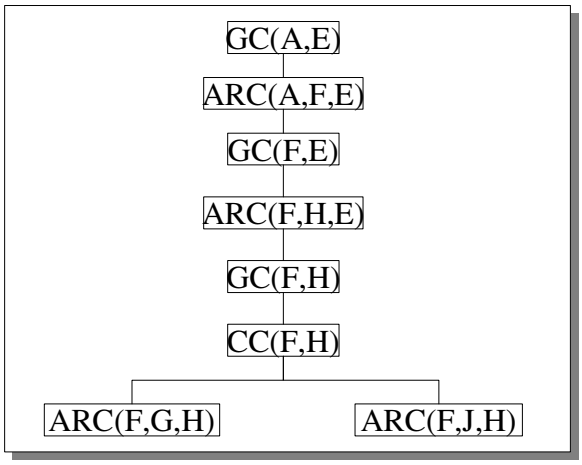

Figure 9 A MTO Channel Construction Tree

As shown in the Figure 9, a MTO channel construction tree can be represented as a hierarchical tree.

Assume that ARC has I number of component, CC has J number of component, and GC has K number of component. Also, assume that the form complexity of each component is one and the form complexity of TCP is assumed zero because the TCP software is already in every nodes. The complexity of ARC(F,G,H), and ARC(F,J,H) is O(I). The complexity of CC(F,H) is O(J+I). The complexity of GC(F,H) is O(K+J+I). The complexity of ARC(F,H,E) is O(K+J+2I). The complexity of GC(F,E) is O(2K+J+2I) complexity. ARC(A,F,E) has O(2K+J+3I) complexity. GC(A,E) has O(3K+J+3I) complexity. Assume that N is a maximum number of I, J and K and D is the depth of the channel construction tree then the form complexity of the channel can represent O(DN). Therefore the channel form complexity is bounded by linear function complexity. In practical situation, the form complexity is dominated by the factor of the depth of channel construction tree because N is usually a fixed small number while the depth of a channel construction tree varies in case by case.

4.4.2 MTO Channel Assembly Time Complexity

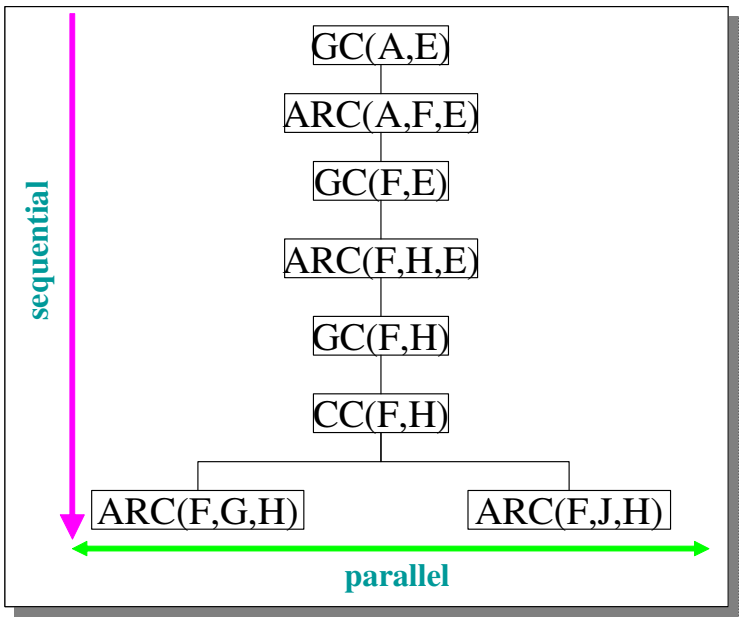

Figure 10 A MTO Channel Assembly Time Complexity

Definition 8: MTO Channel Assembly Time Complexity

The assembly time complexity of a MTO channel C, AssemblyComplexity(C), is defined as $AssemblyComplexity(C) = T(f(P_c)) + MAX(MAX(AssemblyComplexity(C')), MAX(AssemblyComplexity(M_c))) + \gamma$. $T(f(P_c))$ *is a time of planning function of a MTO channel C.* γ *is constant for load and activation time of the MTO channel C.*

The assembly complexity is a time complexity for constructing a MTO channel including its channel planning time and its sub-MTO channel construction time. Independent MTO channels such as ARC(F,G,H) and ARC(F,J,H) in Figure 10 are

constructed concurrently. But a sub-MTO channel has a dependency in its upper-level channel. So, sub-MTO channels are constructed sequentially. The planning time can be done in polynomial time or in exponential time based on its planning function as described in the beginning of section 4.4. A MTO channel returns a construction status message to its upper-level MTO channel after it constructs its components. The upper-level MTO channel needs to wait until its sub-MTO channels are constructed before sending its own construction status message to its upper-level MTO channel. The assembly time includes loading and activation times for components too. The assembly time depends on the size of the component to load and the initiation time of the component. However, if the component size is not abnormally different, in a real environment, the loading and activating times does not vary too much.

Each MTO channel can be constructed independently and concurrently except its own sub-MTO channels. A MTO channel assembly time takes the maximum time of its components that includes sub-MTO channels. Assume that the MTO channel component assembly complexity of ARC, CC, and GC are T_a, T_c, and T_g, and the channel planning time of ARC, CC, and GC are $T(f(P_a))$, $T(f(P_c))$, and $T(f(P_g))$. The assembly complexity of ARC(F,G,H) and ARC(F,J,H) is $O(T_a + T(f(P_a)))$. The assembly complexity of CC(F,H) is $O(T_a + T_c + T(f(P_a)) + T(f(P_c)))$. The assembly complexity of GC(F,H) is $O(T_a + T_c + T_g + T(f(P_a)) + T(f(P_c)) + T(f(P_g)))$. The assembly complexity of ARC(F,H,E) is $O(2T_a + T_c + T_g + 2T(f(P_a)) + T(f(P_c)) + T(f(P_g)))$. The assembly complexity of GC(F,E) is $O(2T_a + T_c + 2T_g + 2T(f(P_a)) + T(f(P_c)) + 2T(f(P_g)))$. The assembly complexity of ARC(A,F,E) is $O(3T_a + T_c + 2T_g + 3T(f(P_a)) + T(f(P_c)) + 2T(f(P_g)))$. The

assembly complexity of GC(A,E) is $O(3T_a + T_c + 3T_g + 3T(f(P_a)) + T(f(P_c)) + 3T(f(P_g)))$. The $O(T_a)$, $O(T_c)$, and $O(T_g)$ is in polynomial time. The loading and activation time for each component is linear, and a MTO channel has linear number of components. So, The channel component's assembly time complexity is O(N) where N is the number of channel components in the channel. So, the assembly complexity of GC(A,E) is $O(N + T(f(P)))$ where P is the most complex planning function among the three channels. If the most complex planning function is a deterministic problem, then the assembly complexity is O(N+P). However, if the most complex planning function is a NP-complete problem, then the channel's construction is performed in non-polynomial time. In practical situation, the assembly time is dominated by the factor of the depth of channel construction tree because N is usually a fixed small number and the planning function is a computation oriented task using a network graph that has limited nodes and links while loading and activation time is an IO oriented task which is a relatively very slow task than computation oriented task.

4.5 Optimization of MTO Channel Construction

As seen in the previous MTO channel formation and assembly complexity calculations, the complexities of MTO channel construction are largely affected by the height of the MTO channel construction tree, i.e. the depth of recursive MTO channel construction. In this section, a MTO channel construction optimization method is

presented to reduce the MTO channel complexity by eliminating the unnecessary MTO channel components.

Definition 9: Optimization of MTO Channel Construction

Rule 1: If MTO channel A has all functions of sub-MTO channel B, remove MTO channel B.

Rule 2: If MTO channel A doesn't use any added function of sub-MTO channel B, remove MTO channel B.

Rule 3: Perform optimization until no optimization is possible.

Because the GC MTO channel does not have any added functionality in data transmission, and the ARC MTO channel has all the functions to combine two ARC MTO channels sequentially, the MTO channel construction tree of Figure 9 can be optimized further.

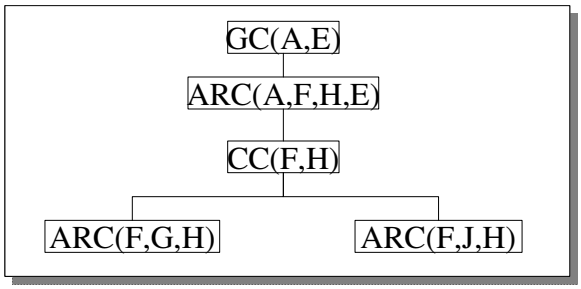

Figure 11 An Optimized MTO Channel Construction Tree

Sequence of optimization of Figure 9 is shown below:

- GC(F,E) and GC(F,H) are removed and reduce tree by Rule 2.
- ARC(F,H,E) are removed and reduce tree by Rule 1.

Figure 11 shows the optimized MTO channel construction tree. The optimized MTO channel construction reduces construction tree depth from 7 to 4.

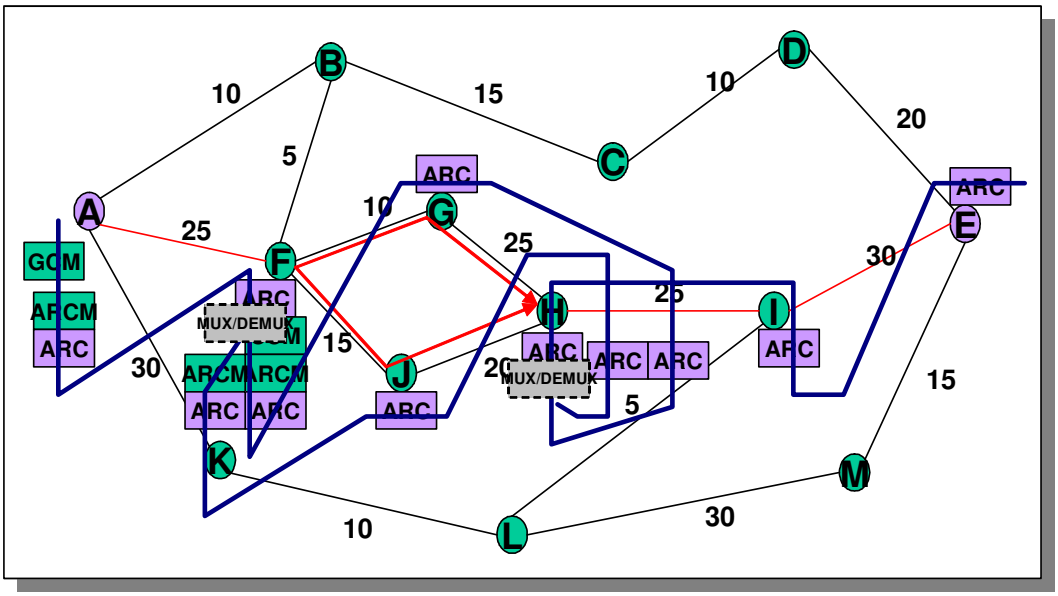

Figure 12 A Final Planned MTO Channel Construction Map

After optimizing the channel, the channel's components are deployed as in Figure 12. It uses an ARC from A to E via F and H, a CC from F to H which uses two ARC MTO channels to connect from F to H concurrently.

4.6 Event Message Path Optimization Issue

An event is generated during a MTO channel's lifetime. An event will be handled by the MTO channel if it has the event handler. If the MTO channel does not have the event handler, then it will propagate the event to the upper MTO channel (super MTO channel which invokes the MTO channel) until the event handler is found or the node operating system will handle the event based on the default configuration. Without event path optimization, this event propagation follows the logical hierarchy of the MTO channel.

The physical deployment of MTO channel components will depend on the component resource requirements, the run time environment, and the placement order requirement. A MTO channel component will be deployed where it is best fit. The physical MTO channel locations may be different from logical MTO channel locations. Therefore an event may be routed through a path which is not the best path to the event delivery. To optimize the event delivery path, the following formula is used to obtain the best path at a given time.

Definition 10: Event Message Path

Location of an event to deliver L_E can be expressed L_E = C.EventHandler(E) if the event is handled in the MTO channel C, or L_E = SuperChannel(C).EventHandler(E) if the message is not handled in the MTO channel C. C.EventHandler(E) is a location of the E vent handler in MTO channel C. SuperChannel(C) is a upper level MTO channel of C.

The discovery of event handler's location is constructed the bottom-up way in coordination with a MTO channel manager. If an event is handled by the MTO channel C's components, the event handle location is set to the channel component location that handles the event. If it is not handled by itself, the event will propagate to its upper layer via its channel manager. So, the location of the event handler should be retrieved from the upper layer (i.e. super MTO channel.)

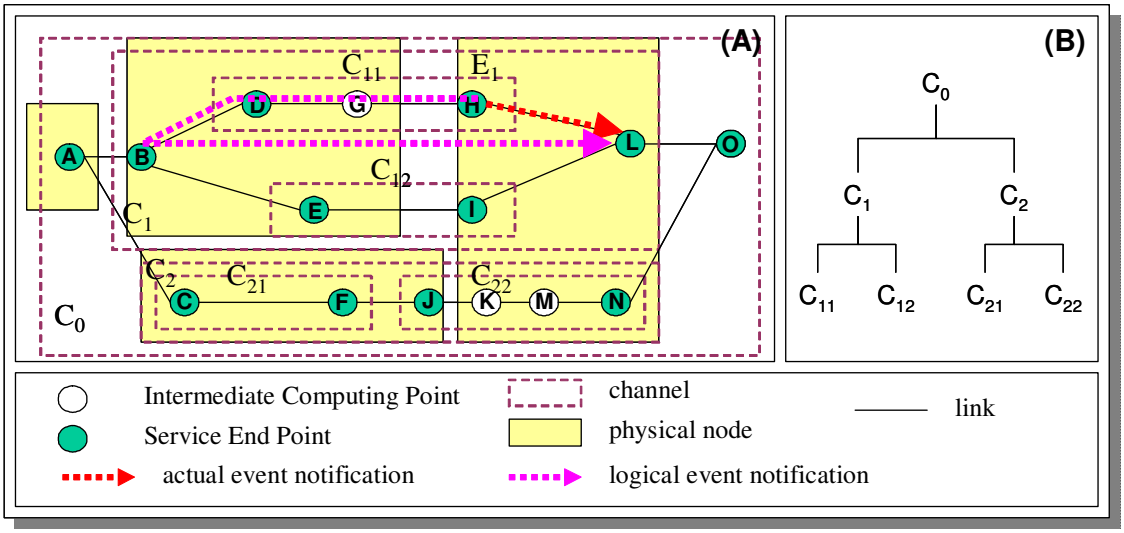

Figure 13 Logical and Physical Event Notification Flow

Figure 13 (a) shows the event notification path of E_1, and Figure 13 (b) shows the MTO channel hierarch. The C_0 has two sub-MTO channels C_1 and C_2. In C_1, it has two sub-MTO channels C_{11} and C_{12}. The MTO channel C_1 has MTO channel manager in POP B. The MTO channel C_{11} has MTO channel manager in POP D. Event E_1 is produced in MTO channel C_{11}'s POP H and the event E_1 is handled in C_1's POP L. Without event

message optimization, the event E_1 is notified to the MTO channel manager in POP D. The MTO channel manager of C_{11} then forwards the event E_1 to its upper level MTO channel's manager which is located in POP B because it does not have an event handler of E_1. The C_1's MTO channel manager then forwards the E_1 to the handler located in POP L. The pink line is the logical path of the Event E_1 notification. It is following the MTO channel C_{11}'s path and reported to the MTO channel C_1 (POP B) and forwarded to the handler in POP L. However, after the event location discovery process, it redirects the event notification to POP L, where the E_1 is handled.

All events also have their default event handler location, the MTO channel manager's location. When an event handler doesn't respond because of link down, network congestion, unavailable system, and so on, the event will be delivered to the default event handler which is included in a network operating system.

4.7 Analogy with TCP connection

A TCP connection is a traditional transporting object in between communication entities. A definition of a TCP channel can be defined C_{TCP} = ({TCP_stack }, {IP}, {IP_{src}, IP_{dst}}, {}, {closed, listen, syn_rcvd, syn_sent, established, FIN_WAIT_1, FIN_WAIT_2, closing, time_wait, close_wait, last_ack}). It uses IP protocol as its sub-MTO channel. One of the noticeable things in these definitions is an event set which is an empty set. TCP does not produce event. It handles all the generated events such as SYN, ACK,

DATA, and FIN by itself, and changes its status rather than produces events. Therefore, a library function checks the status variables to check if the event occurred.

The TCP protocol stack is the only TCP connection's component which is preinstalled at both communication entities. The TCP protocol stack performs manager and service roles. The TCP protocol stack works as an interface point to the upper layer or application (manager). It also handles actual data transmission as a service component. However, it does not need to gather network information actively like Scout in a MTO channel.

The construction of a TCP connection is undertaken using IP protocols to send SYN and ACK to establish a connection. All required software components are preinstalled in both communication end points and intermediate nodes also have IP protocols to forward the packets. Because the required software is preinstalled in all involved nodes, the construction of a TCP connection does not required dynamic installation. It also does not require using path discovery phase to install a channel component. Destruction of a TCP connection simply releases connection information; it does not unload the protocol components after termination of the message exchange.

TCP has two points as point of presence; source and destination. It has one connection from source to destination with given ports. TCP also maintains its own states: closed, listen, syn_rcvd, syn_sent, established, FIN_WAIT_1, FIN_WAIT_2, closing, time_wait, close_wait, last_ack.

Table 24 Information Metric of TCP from A to E

V_{EP}	IP	IP(A)	IP(B)	IP(E)
	Role	S	F	R
	Label	A	B	E
M_{ROT}	A	*		*
	E	*		*
G_{BASE} and M_{QON}	A		10	
	B	10		

The Information Metric of TCP is shown in Table 24. TCP has partial network metric because it only can have a next hop to the destination. The M_{ROT} is unknown because it is not a guaranteed service. It does not have DMAP and M_{QOS}; all the required software components are already installed, and it cannot predict the quality-of-service.

CHAPTER 5

Active MTO Channel System Architecture

5.1 Overview of Active MTO Channel System

5.1.1 Active MTO Channel System Model

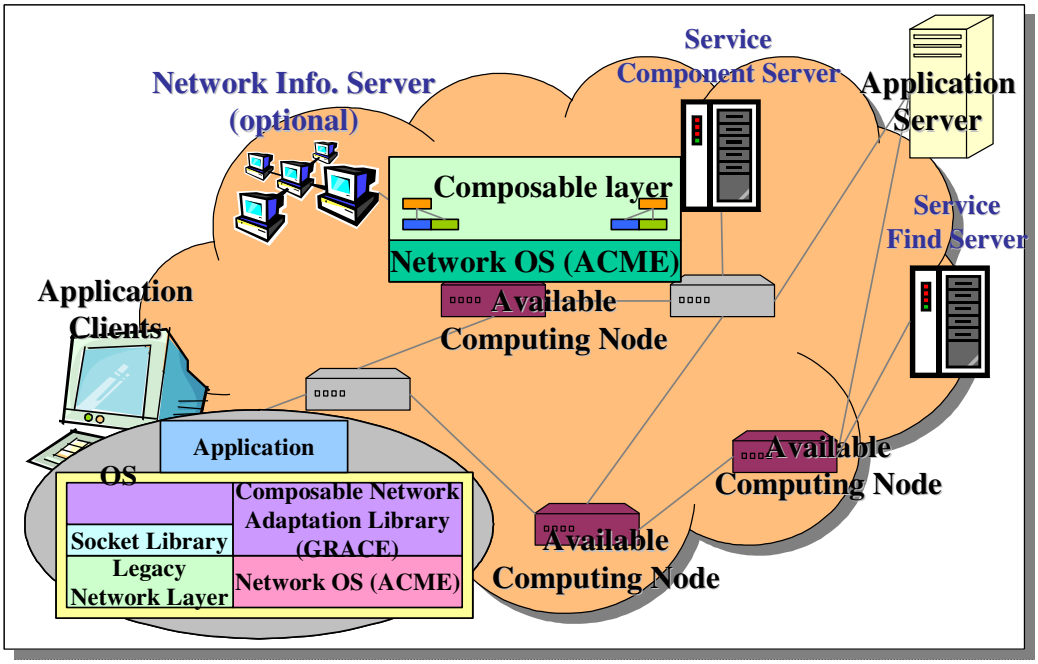

Figure 14 An Active MTO Channel System Model

Figure 14 depicts an active MTO channel system model. It consists of applications, active MTO channel system library (GRACE), network OS (ACME), service find server(s), service component server(s), and network information server(s). Applications use a special library called the general active channel constructor extension (GRACE) to

use a MTO channel. The GRACE is very close to the Berkeley socket library. Active channel management enforcer (ACME) is an added component in a network node operating system to support dynamic composition of the MTO channel in each active node as well as in application nodes. A service find server is used to bind the user's requirement to a specific MTO channel service. A service component server is a repository of MTO channel's components and serves for dynamic component loading. A network information server is an optional server to provide network wide information such as the network connection map and physical resources of the network/system in the network which might be needed by manager components to know about the network topology and capacity metric.

5.1.2 Active MTO Channel System Node OS Service Layer Architecture

The Active MTO Channel System Architecture is divided into three tiers: application subscriber layer, compose-able service layer, and enhanced network layer. Figure 15 shows the layers.

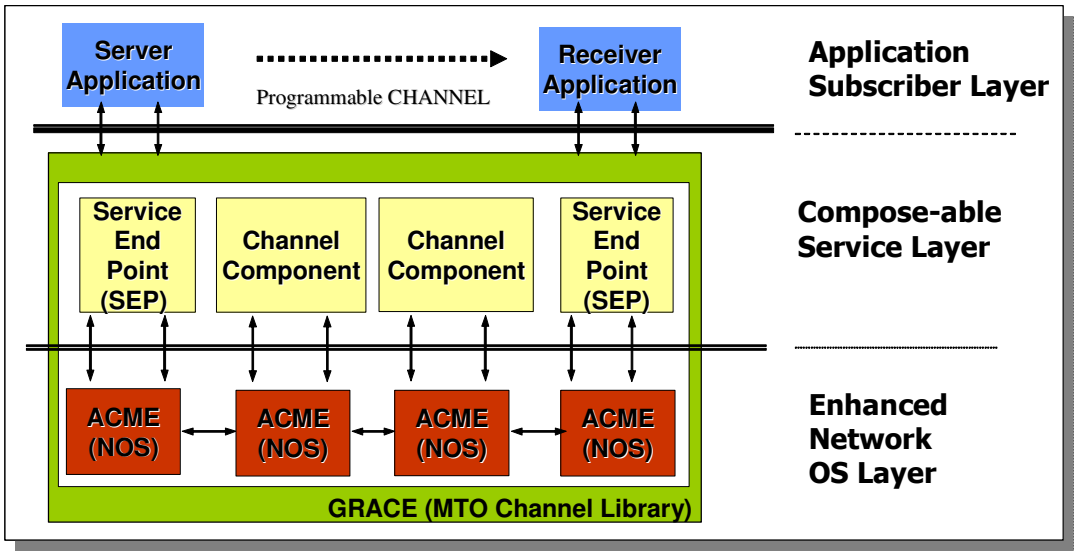

Figure 15 The Three Tiers of System Architecture in Active MTO Channel System Formalism

The first is the *application subscriber layer*. More specifically, the network independent part of the domain routines resides in this tier. The second is the *composeable service layer*. The components in this layer are programmable. However; they execute strictly under the control of the enhanced OS layer and help in the local state's dependent application. The third is the *enhanced network OS layer* which houses the service components that are loaded and executed. It bridges between the services and applications.

5.1.3 Application Subscriber Layer

Many parties participate in the construction and maintenance of a dynamic service composition. The process is not a simple task but it does not mean to be complex to use.

The service construction needs to be easy for an application. Of the three layers, the application subscriber layer has the best knowledge about what are the applications/users requirements. The application should remain independent from the service layer, but it should have the means to reflect its requirements to a service and to access the status of the service. The supplemented MTO channel library, GRACE, supports uniform interfaces to the application and hides the complexity of the MTO channel construction from the application developer.

Current Socket APIs are one of the most successful interfaces for network communication. Only a small set of APIs is necessary to set up and use a network service. The APIs are independent from the operating systems, physical hardware, and underlying protocols. All the management of a connection for the socket is hidden from the application. Because of their simplicity, application programmers love to use socket interfaces. When new protocols are developed, an augmented set of options are included in the socket APIs. These services options should be defined when a socket is created. In current socket API these services are static and the options are not flexible. The required functionality should pre-exist in the network layer before it can be used. Also, for some services, each communicating party involved in the transmission should support the option and underlying functionality. So, the available service is limited further from the function set given from the one entity.

The active MTO channel system is supported by the new GRACE application user interface. It has been designed as the socket library. However, the underlying functionalities of the GRACE are quite different from the legacy socket library. It

borrows the simplicity and forms from the socket library. The GRACE supports dynamic MTO channel binding. Therefore, the GRACE's role is quite different from that of the general socket library.

The GRACE is used as an end point programming interface (EPPI) by applications to access MTO channels. The GRACE application user interfaces are shown in Table 25. Using GRACE, an application can launch any MTO channel.

5.1.4 Compose-able Service Layer

A compose-able service layer is a layer where the custom MTO channel resides. In this layer MTO channel components are loaded, bond, and are executed. A MTO channel may have more than one MTO channel component. When a MTO channel component(s) is required, a node operating system (ACME) loads the component and executes it in this MTO channel layer space. This layer is managed by the ACME. ACME supports basic functionalities, i.e. load, execute, terminate, and unload components, and the scheduling and managing of system resources. The requests of MTO channel load/execute/terminate/unload are invoked by a MTO channel manager which is a representative component in a MTO channel, or an application library at initial MTO channel set up time. ACME may actively manage MTO channel layer, i.e. purging the unused MTO channel components, but a MTO channel manager takes responsibility for the management of its MTO channel components. Communication in between the MTO channel components is predefined by a MTO channel designer. However, a MTO channel

can use other MTO channels to perform or augment its function. The communication between a MTO channel and its sub-MTO channel takes place the same way that the application library calls a MTO channel. In this fashion, the MTO channel can construct complicated functions using the other MTO channels while maintaining the simplicity of its interfaces.

5.1.5 Enhanced Network OS Layer

The network layer should be augmented for the dynamic MTO channel load and execution. Legacy networks do not require load and execution because all the functions in such networks are pre-loaded when the operating system is loaded. Even though operating systems use dynamic libraries to load and unload a set of network components in run time, the function set which is available to the system remains the same.

The basic functions of the enhanced network OS layer are load, unload, execute, and terminate with regard to a MTO channel component. However, for system integrity, the enhanced network OS layer should limit the usage of the system using policies. The enhanced network OS layer can limit the MTO channel components that can be executed in the node, as well as the usage of system resources, and can validate the authority of the requests, and so on. The network operating system (ACME) in this dissertation is used to indicate the enhanced network OS layer.

5.2 Application Programming Interfaces

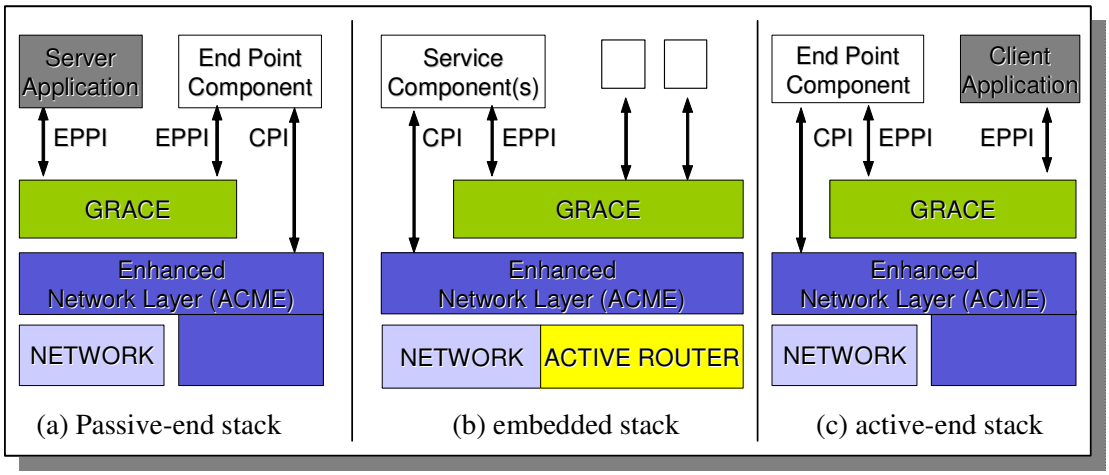

Figure 16 Network Services Layers for Dynamic Service System Construction

Programming Interface: Unlike classical channels, the network OS provide two interfaces. First is the end point programming interface (EPPI) to the application (source(s) and sink(s)) or to MTO channel to request and run a MTO channel. The other is the component programming interface (CPI) for MTO channel components to execute and co-ordinate their functions. The general active channel constructor extension (GRACE) supports EPPI. The active channel management enforcer (ACME) supports CPI. It is possible that a MTO channel component uses EPPI for managing its sub-MTO channel. An application may directly call ACME APIs also, but it is not recommended.

Services: Below the interface, the following new network layer services are now required for a MTO channel: (a) MTO channel installation service (b) intra-channel-component communication service, and (c) network state exchange service. All MTO channel components are categorized in three different component types. A manager,

which represents the MTO channel and takes responsibility of the MTO channel service usually located at the actuator end of the connection. The service components, which are deployed in the intermediate nodes, perform service on the contents and deliver them to the next point. Scout components are optional which are deployed and perform information gathering or processing before the MTO channel is fully installed. The figure above shows the service stacks in three positions. The junction nodes use EPPI, but no application component runs on them. However, since, the junction points can be a typical active router, instead of relying on general OS, an enhanced network layer is created on top of the router. The enhanced network layer identifies dynamic MTO channel communication from regular routing operation and when dynamic MTO channel data arrives, it diverts the dynamic MTO channel packet towards proper components. The enhanced network layer also allocates the CPU and memory resources among the competing MTO channel components and acts as a MTO channel component scheduler. To support a composite MTO channel, the active MTO channel system supports following groups of APIs.

A MTO channel construction is initiated by an application by calling a GSocket library function. The general active channel constructor extension (GRACE) provides similar functionality to the socket library while it is hiding the internal differences of the MTO channel architecture. The application user interfaces for the construction of a MTO channel is shown in Figure 17.

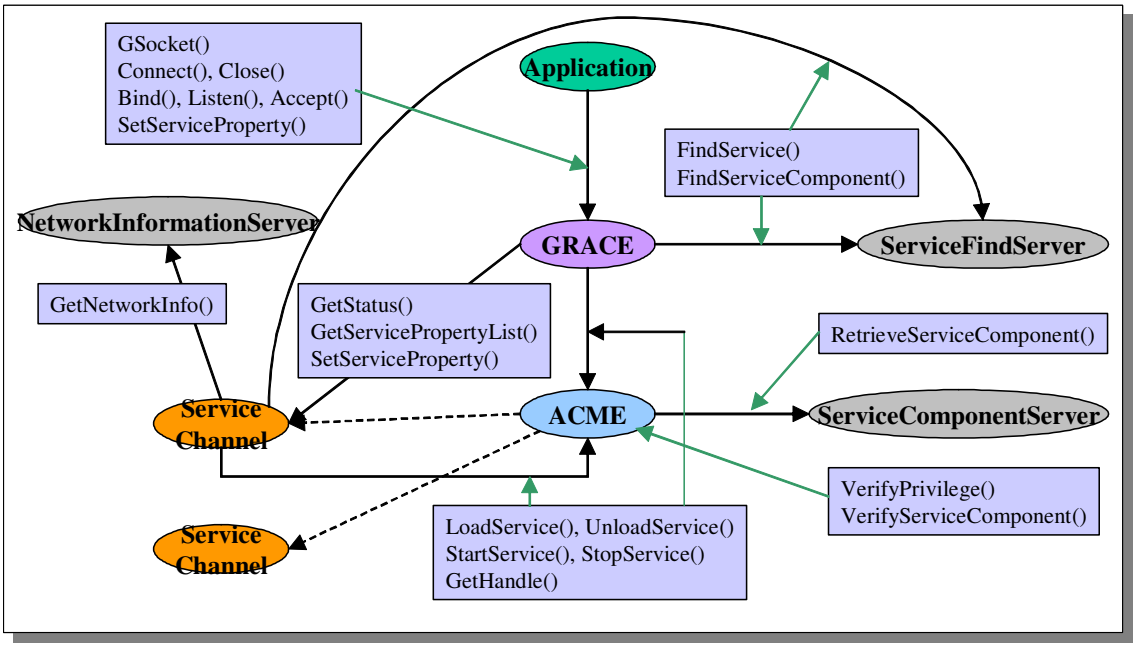

Figure 17 Application Programming Interfaces for a MTO Channel Construction

There are six entities participating in the service constructions. Table 25 shows ACME interfaces. ACME interfaces are used as CPI to service components. It supports load/unload/start/stop of a service, verification of privilege and a service component's integrity as well as the retrieval of a service handler.

Table 25 Active Channel Management Enforcer (ACME: Network OS) APIs

Interface	Description
int LoadService(servicename)	Load a service
int UnloadService(servicename)	Unload a service
servicehandle StartService(servicename)	Start a service
int StopService(servicehandle)	Stop a service
servicehandle GetHandle(servicename, servicedescription)	Get a handler a service by name
int VerifyPrivilege(userid, operation, targetobject)	Check and verify privilege of a user for the operation
int VerifyServiceComponent(servicecomponentname)	Check and verify a service component's integrity

Table 26 Service Component APIs

Interface	Description
status GetStatus()	Retrieve status information of the service
servicepropertylist GetServicePropertyList()	Return a service property list
int SetServiceProperty(serviceproperty, value)	Assign a service property value
int Send(databuffer, size)	Send contents using the service
int Receive(databuffer, size)	Receive contents via the service

The service components should support five APIs by defaults as shown in Table 26. These five application programming interfaces should be implemented in all service components. They have status information retrieval, property handling, and data communication functions.

Table 27 General Active channel Constructor Extension (GRACE) APIs

Interface	Description
sockethandle GSocket(service)	Define a GRACE socket
int Connect(sockethandle)	Connect a service entity
int Close(sockethandle)	Close a service
int Bind(sockethandle)	Binding a GRACE socket
int Listen(sockethandle)	Listening a GRACE socket connection
sockethandle Accept(sockethandle)	Accepting GRACE socket connection
servicepropertylist GetServicePropertyList(sockethandle)	Retrieve a service property list
int SetServiceProperty(sockethandle, serviceproperty, value)	Assign a service property
int Send(sockethandle, buffer, size)	Send contents using the GRACE socket
int Receive(sockethandle, buffer, size)	Receive contents via the GRACE socket

GRACE application programming interfaces are used by applications and a MTO channel component to create a sub-MTO channel. Table 27 shows the interfaces. It is designed as close as the Berkeley Socket interface to easy migration of a socket programmer to the new MTO channel programming yet enough to support the exploration of a custom MTO channel service.

Table 28 Service and Network Information APIs

Entity	Interface	Description
Service Find Server	servicename FindService(service)	Retrieve a service name
	servicecomponentname FindServiceComponent(servicename)	Retrieve a service component name
Service Component Server	int GetServiceComponent(servicecomponentname)	Load a service component
Network Information Server	NetworkMap GetNetworkInfo(networkinfoname, buffer)	Retrieve a network information

The service find server, service component server, and network information server interfaces are shown in Table 28. The service find server interfaces are used by GRACE and service component to find a service that satisfies the specification as well as the functionalities. The service component server interfaces are used by GRACE and a MTO channel manager to load a service component. The network information server may be used optionally. The network information server sends network information as a return to the GetNetworkInfo() interface call.

5.3 Single MTO Channel Construction

The MTO channel construction sequence is explained with an example of a basic MTO channel service.

A basic MTO channel is a simple forwarding MTO channel service. It consists of BC_MGR, BC_SRC, BC_CTR, and BC_DST. BC_MGR is a manager component of the MTO channel. It will charge for the rest of the MTO channel construction after it is

invoked by the system. BC_SRC and BC_DST are source and destination components which send and receive data from/to the application it connects. BC_CTR is a MTO channel component deployed in the path in between the applications.

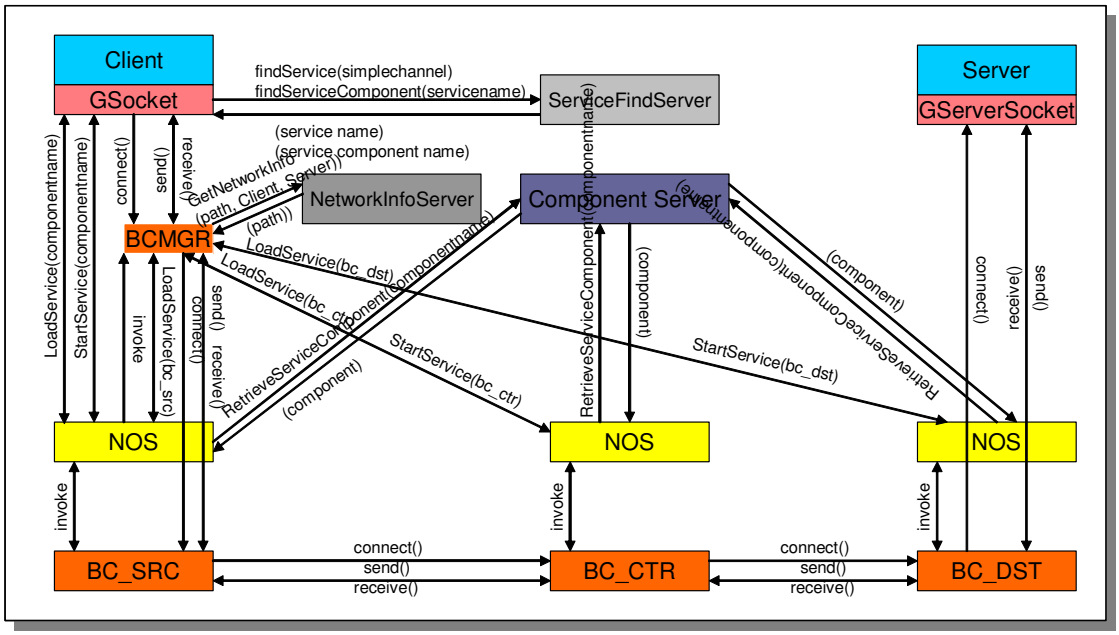

Figure 18 A Single MTO Channel Construction Sequences

Figure 18 describes a single MTO channel construction sequence using a basic MTO channel service. A client application requests a simple MTO channel service from a GRACE. The active channel system library, GRACE, contacts a service find server via findService(). A service find server returns a proper service name, i.e. basic MTO channel. The GRACE also requests to find service component using a findServiceComponent() function call. When the GRACE has the service component name, it requests load of a MTO channel component, BC_MGR, to the node operating system, ACME. The ACME checks its repository for the requested MTO channel component. If the ACME does not have the component, it contacts its service component

server to load the component. After the component is loaded, the GRACE invokes the component (a manager component) to initiate a MTO channel construction. The invoked MTO channel manager starts its predefined sequences with given information from the application, e.g. end points, required bandwidth, et cetera. The MTO channel manager uses the network information server or uses Scout components to discover the network map. With the network map, the MTO channel manager decides the proper locations for its other components, in this case BC_SRC, BC_CTR, and BC_DST. The MTO channel manager requests ACME(s) for loading other components and requests to invoke them. The other components start to connect other service components after they are initiated. The location information of other components is sent to a service component by a MTO channel manager. After all, the MTO channel components are connected, the manager signals to the application that the MTO channel construction is finished and ready to use. The application sends data to the MTO channel after it receives the ready signal from the MTO channel.

5.4 Multi-level MTO Channel Construction

A multi-level MTO channel construction is similar to a single MTO channel construction. It has an added sub-MTO channel construction phase and a MTO channel optimization phase when a sub-MTO channel is required for supporting the service requirements. A MTO channel manager requests a finding service from a service find server that is the same request as in the application does. A MTO channel manager uses

service MTO channel name returned from the service find server to invoke the sub-MTO channel. The MTO channel manager also sends parameters to construct sub-MTO channels. The invoked sub-MTO channel manager deploys and invokes its sub-MTO channels and components independently from upper level MTO channel construction with given parameters. However, when a sub-MTO channel manager encounters an error during its MTO channel construction, it generates an event and notifies the upper-level MTO channel manager to coordinate the error handling. The upper-level MTO channel manager re-organizes its sub-MTO channel or components, or it makes an error event and notifies the error to its upper-level MTO channel manager.

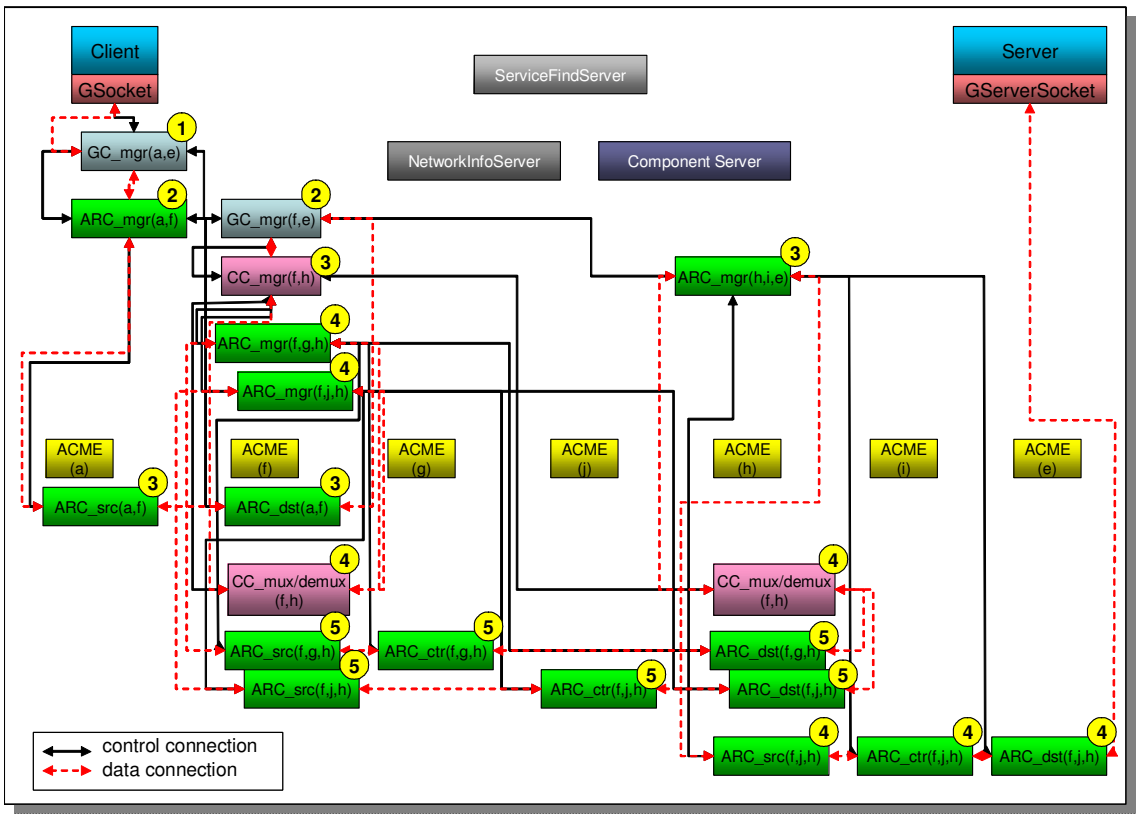

Figure 19 A Multi-level MTO Channel Construction Sequences

Figure 19 presents signaling among the MTO channel construction systems of a generic MTO channel from POP a to POP e which is shown in Figure 11. A client request for a GC from a to e initiates the construction. GSocket() requests to find the proper MTO channel service which has bandwidth indication features, in this case GC. Using the name returned from a service find server. GSocket() contacts ACME for loading and initiating the GC MTO channel service. The ACME loads the GC MTO channel manager (GC_mgr) and invokes it. GC_mgr requests network information from a network information server. After GC_mgr discovers the deployment plan using its planning algorithm, it requests to find a MTO channel service which has alternative path construction features. The name of the altered routing channel (ARC) is returned from the Service Find Server, and the GC_MRG uses this MTO channel name to request loading and initiating the MTO channel service manager (ARC_mgr) for a to f. It also uses the GC to construct a MTO channel for f to e. ARC(a,f) constructs itself by loading and initiating its other components (ARC_src, ARC_dst). GC(f,e) uses a concurrent MTO channel to fill up from f to h and uses a ARC(h,i,e). CC(f,h) finding altered routing service by query to a service find server and uses ARC(f,g,h) and ARC(f,j,h) as its sub-MTO channels. The CC(f,h) also completes its construction by loading its own components, CC_mux/demux, in f and h then connects the ARC(f,g,h) and ARC(f,j,h)'s SEPs to its CC_mux/demux. After CC(f,h) and ARC(h,i,e) is constructed, the GC(f,e) connects those two sub-MTO channels. The GC(a,e) connects ARC(a,f) and GC(f,e) to

complete its construction. Finally, GC(a,e) notifies the application of of its readiness, and the application initiates communication.

5.5 Meta Information Language

5.5.1 Brief description of MIL

A meta information description language (MIL) is required for language independent interface design in building network centric MTO channel system. MIL is used to describe MTO channel component interfaces, MTO channel component manuals, network information, computational information, and user requirements. With MIL, a MIL parser generates an automated manual for MTO channel component developers and end users.

Some of the MILs, the one that describes network information for instance, should be interpreted in run-time. Therefore, the MIL should be a simple language to interpret in run-time.

5.5.2 MIL Grammar

The MIL grammar is shown in the following figures.

```
[api] = [function]+
[function] = <function>
             <name>string</name>
             (<desc>string</desc>)
             (<param>parameter</param>)*
             (<return>parameter</return>)
             </function>
```

Figure 20 The MIL Grammar for API Description

Figure 20 describes the MIL grammar for API description. API is composition of functions. Each function starts with a <function> tag and ends with </function>. In each function, the name of the function is a mandatory field while description, parameters, and return value are optional.

```
[PMAP] = <pmap>
             <name>string</name>
             (<desc>string</desc>)
             <origin>[location]</origin>
             <topology>[topology]</topology>
             ([component])+
             <order><netorder>[orders]</netorder>
             <exeroder>[orders]</exeorder></order>
             </pmap>
[location] = <url>string</url>
[topology] = tree | linier | star | ring | weighted graph
[component] = <component>
             <name>string</name>
             (<desc>string</desc>)
             <location>( * | [node] | on [component].set) </location>
             (<invoke>( * | [time] )</invoke>
             </component>
[orders] = ((<seq>(component)+</seq>) | (<concur>(component)+</concur>))+
```

Figure 21 A Placement Map MIL Grammar

A placement map MIL grammar is shown in Figure 21. This grammar is used by the MIL parser and generates MTO channel component placement information.

5.5.3 MIL parsing and output generation

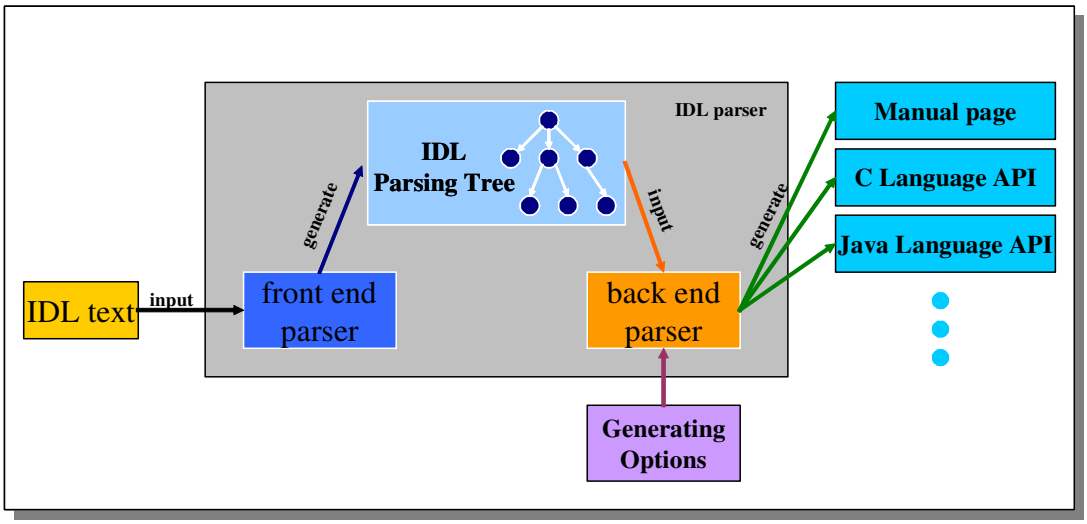

Figure 22 A MIL Parsing and Output Generation Architecture

The MIL parser has two components, the front end parser and the back end parser. The front end parser generates a parse tree based on an input MIL text. The generated parse tree is an internal representation of the MIL description. The back end parser takes the generated parse tree and produces a proper output format indicated by generation options. Different language output requires only a different back end parser.

5.5.4 Parsing tree structure

The parsing tree consists of three properties, name, value and attributes. The name property contains the node name in a parsing tree. The value property has content value

for the name. The attribute property has an attribute of the name and value set. Each node may have child node(s).

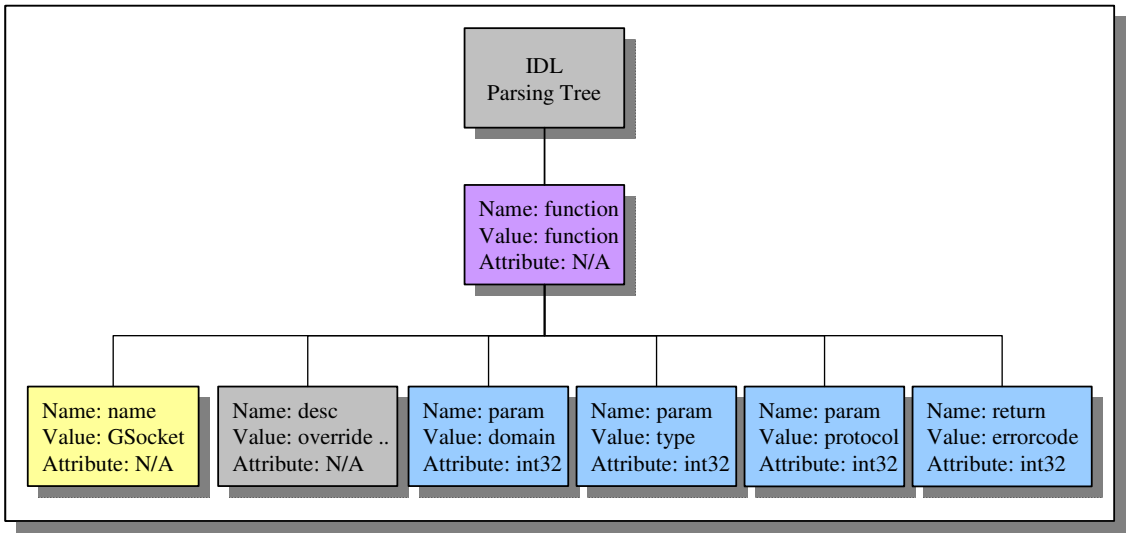

Figure 23 A MIL Parsing Tree Structure

Figure 23 represents an example of the MIL parsing tree structure for the GSocket function call described in Figure 24.

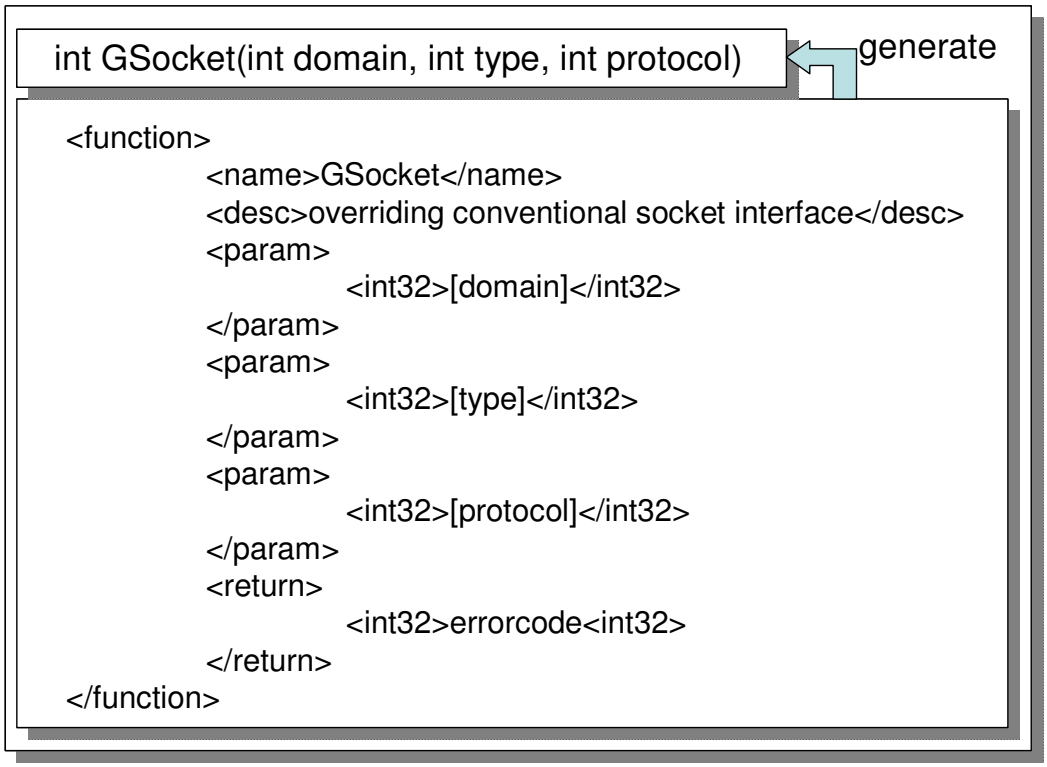

Figure 24 A MIL Description for a GSocket Function Call

The MIL parser reads the description and generated an output as shown in Figure 24, in this case a back end parser is the C programming language parser.

```
<PMAP>
 <name>Altered routing channel</name>
 <desc>
   <author>author</author>
 </desc>
 <origin>
   <url>https://medianet.kent.edu/capsules/*.cap</url>
 </origin>
 <topology>[tree]</topology>
 <component>
   <name>altc</name>
   <desc>ARC channel Manager</desc>
   <location>*</location>
   <invoke>*</invoke>
 </component>
<component>
   <name>altc_src</name>
   <desc>ARC actuator</desc>
   <location>Actuation End Point</location>
 </component>
<component>
   <name>altc_ctr</name>
   <desc>ARC forwarder</desc>
   <location><each><path></each></location>
 </component>
 <component>
   <name>altc_dst</name>
   <desc>ARC audience</desc>
   <location>Audience End Point</location>
 </component>
 <order>
   <netorder>
     <seq>[altc][altc_src][altc_ctr][altc_dst]</seq>
   </netorder>
   <exeorder>
     <seq>[altc][altc_src][altc_ctr][altc_dst]</seq>
   </exeorder>
 </order>
</PMAP>
```

Figure 25 A PMAP for Altered Routing Channel

The PMAP for an altered routing MTO channel is described in Figure 25. This description also has enough information such as the number of software components and who ordered them. With this meta information, the run-time system determine which place a MTO channel component can be located.

CHAPTER 6

System Visualization

Though traditional networking research has ignored visualization, the monitoring and management of complex distributed systems is becoming critical for high-performance distributed computing. However, monitoring an active distributed system such as Grid, the active application or the Internet content services have several serious obstacles to overcome. The first set of complexity evolves from the scale, dynamism and versatility requirements. An additional challenge arises from autonomous ownership of the Internet systems. It is further complicated by the hierarchical and multi-party nature of the netcentric systems development pathway. During run-time, a sound message management principle becomes very important otherwise, a potentially huge number of status messages can result in a serious performance drag.

6.1 Related Researches on Visualization

Recently, there have been few pioneering works in the area of Grid visualization. Tierney, et al. [4][5] suggested an agent based monitoring system to automate the execution of monitoring sensors and the collection of event data in a Grid environment. They use a direct connection between a producer and a consumer to reduce communication traffic.

Waheed, et al. [8] developed monitoring infrastructure to share monitored data using common APIs. The infrastructure is built on three basic components; sensors, actuators, and a Grid event service; at the top of those basic components, they built a layered monitoring system. Another layer-based visualization system was suggested by Bonnassieux, et al. [6]. They offered a flexible presentation layer in a huge and heterogeneous environment. It provides a simple, autonomous and extensible model that enables the visualization of any level of abstraction using a hierarchical view model of resources status, with propagation of monitoring status up to the top of the tree view. The gathered information for monitoring can also be used for system management. Reed, et al. [7] suggest using system monitoring results for adaptive control to improve system reliability. The system uses diskless check-pointing, which enables more frequent checkpoints by redundantly saving check-pointed data in memory, and low-cost mechanisms to capture data for failure prediction, which enables the creation of dynamic schemes for improved application resilience.

6.2 Visualization Architecture Requirements

One of the major challenges that differentiate netcentric systems from traditional modular distributed software is the fact that the concept of internet autonomous systems (that separate the network from the Internet) also extends to the software systems. This hierarchically dependent multiparty involvement extends to both the development process of compose-able services as well as to the runtime service ownership. Clearly,

these systems are not built as a one simple big program. Rather, they are built with several independent system components running on multiple computing systems. Each system component is also composed with several sub-components and distributed among multiple computing systems. Also unique is the fact that, quite often these are developed under multiple autonomous service authorships, and deployed and managed under multiple service ownerships. Because of such a nature, system monitoring and controlling get considerably more difficult and complex. In addition, the current trend whereby network based sub-systems and components have to go through frequent modification for the newly included or upgraded components makes the overall task even less manageable. As a result, the system management and monitoring software encounters difficulties to visualize the whole system across the participating computing systems and services, and sometimes the software is faced with disparity between system status reports and control messages and their representation in the system. However, the same complexity makes monitoring and visualization of the process nevertheless more critical. Therefore, there should be support for autonomous modular visualization.

6.3 Other Features of Visualization

The following architectural features are offered to overcome the challenges of the netcentric system's visualization:

- Controllability of message flow.
- Adaptation of system viewer's perspectives.

- Use of meta information for interpretation of current system information.

As the system gets bigger, the generated system status messages also grow. Without controllability of a system message flow, a system is easily overwhelmed by the generated status messages and cannot deliver important information.

The system status representation should be useful enough to produce multiple points of views. A user requires different perspective views depending on a user's interests at a given moment. Representing the same system information in various ways will increase a user's focus on his/her interests.

The separation of system information data and its structures by using meta information makes it easy to upgrade system components while the system is running and verifying message information as well. Using the meta information also leverages the automation of system information representation. A system can generate various target representations by dynamically interpreting message data with its meta information.

A visualization schema is built on the powerful process description language of Petri Net. A Petri Net is a graphical and mathematical modeling tool which consists of places, transitions, and arcs that connect them. [16] It is a powerful tool for modeling systems that are concurrent, asynchronous, distributed, parallel, nondeterministic, and stochastic. It is well suited to describe a system's status and its transition. Recent proposal of Petri Net Markup Language (PNML) is pushing Petri Net language to a more interchangeable format for system modeling. [15].

The proposed framework is particularly suitable for monitoring the lifecycle of loosely coupled and scalable complex multiparty active systems. The developed formalism allows the sub-system to maintain its own status and control messages within it. A sub-system, when used as a part of a high level composed system, can further report its status and control messages to its upper level system. Furthermore, each level supports several reporting and message propagation modes to allow performance tuning. The time, type, and content of messages are decided initially by the system designer. However, these default behaviors can be overridden by a system operator or an administrator at run-time. A privileged user can freely control and monitor the system status using a flexibly configurable multi-view visualization system from any authorized terminal.

6.4 Visualization of Architecture

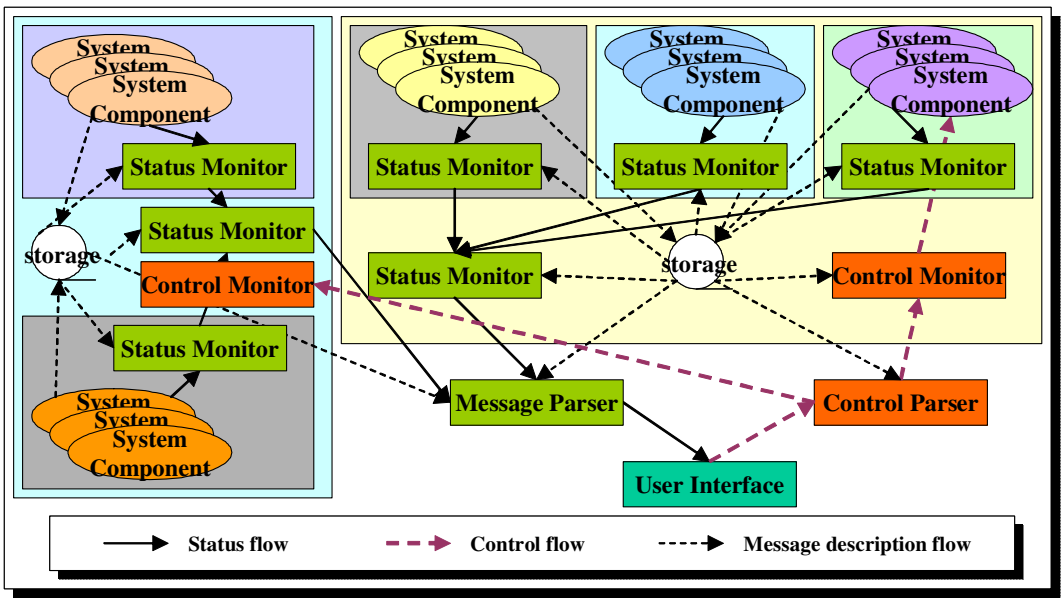

Figure 26 The Visualization System Architecture

Figure 26 describes the symbolic representation of the visualization system architecture. The monitoring components of a service are run on ACME. They are deployed and executed on ACME as a part of a service construction. A status monitor (SM) processes the status message of a sub-system. A SM stores status message structure descriptions and delivers or saves status messages of the sub-system. A control monitor (CM) handles control messages. A CM is added in a sub-system when the sub-system supports a control mechanism from outside of the system. Initiation and execution of a monitor is coordinated by sub-system management software.

6.5 Dynamic Message Binding and Interpretation

Based on the service design considerations, the visualization system should support 1) dynamic interpretation of the system status messages, 2) seamless navigation through layer abstraction and the visualization of the given layer of a system, 3) uniform method of visualization at all levels. The meaning of a status message is represented in a well formed status structure description language and is gathered by a status monitoring system. When a new system component is developed, the descriptions of its status message structures and the descriptions of its state diagram are supplied by the developer together with the component. As per this template, a status monitoring system and a visualization system dynamically bind and interpret the meaning of a status message with the given description.

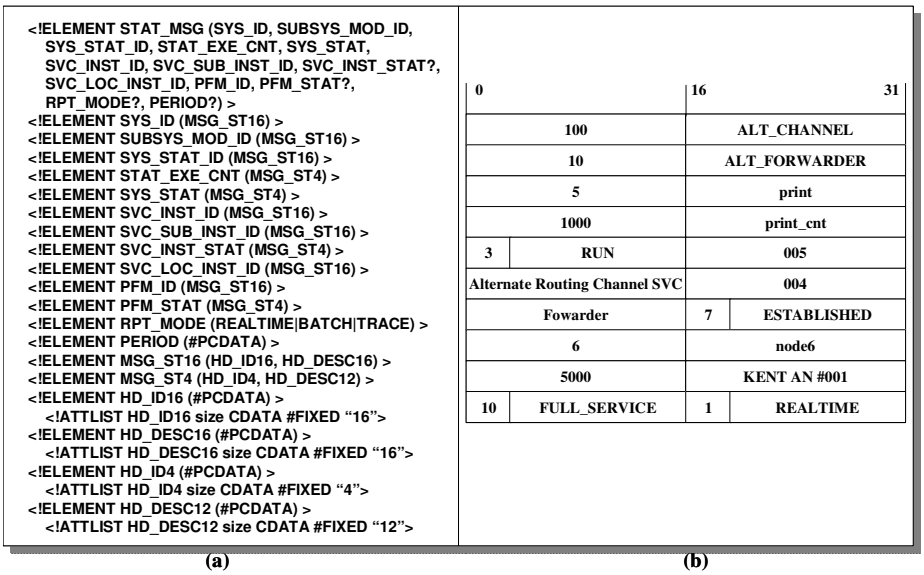

Figure 27 A Status Message Structure and a Status Message

The visualizer integrates with a code server-based hierarchical service deployment framework. Each system can have isomorphic sub-systems and/or code components. Each time a system is installed (i.e. all of its sub-systems are launched), a hypothetical state monitor is assumed to be concurrently instantiated. A set of messages is generated towards this state monitor in the sub-system's leader component. A visualization system can use a subset of the messages to present various perspectives on the system. The key challenge here is that these messages should carry enough information to identify itself with respect to the various perspective frameworks within which an active service operates along with the actual status information. Figure 27 shows an example of status message structure and a status message. Below a tri-partite identifier system is provided. This message system encodes the fields in its messages (i) system identifier, (ii)

subsystem component identifier, (iii) system state identifier, (iv) state execution count, (v) system status, (vi) service instance identifier, (vii) service subsystem instance identifier, (viii) service instance status, (ix) service location instance identifier, (x) platform identifier, and (xi) platform status. The primary state identifier (set i-iii) is assigned by the programmer who has coded the active components. This identifier set has to be *hierarchically unique* within a specific version of specific software. The identifier set (vi-vii) is to be assigned by the active service administration system (such as EEs/ ANETDs) while installing and initializing instances of the service at each instantiating of loaded components. Again, these identifier sets have to be *hierarchically unique* within the service administration domain. The last identifier x is to be supplied by the active node owner. It is assigned when a node joins an active network domain. The status information iv and v is computed by the code components at run time and thus its value is designed by the programmer. The service instance status information viii, if any, is passed on to the monitor messaging agents by the service administration local agent (such as node EE). The status information xi, if any, is set by the local node administrator during the period the service is running. The monitor messaging system collects and composes the messages prior to generating the messages. Messages can contain control flags to control the mode of reporting and even to filter the content to tune performance. The system allows three reporting modes (i) REAL-TIME, (ii) BATCH, (iii) TRACE-ONLY. In real-time mode the monitor messages are generated and sent when the code executes through the state points. In BATCH-ONLY mode the messages are generated in real-time but forwarded periodically in batch. The period is decided by a PERIOD field. The mode

feature only modifies the time of sending the monitor messages but do not affect their content. Three flags are further used to negotiate filtering the three status fields in the messages. In every message sent by the monitor messaging agent, the flags are set according to the current value of these flags. A set of control messages can be potentially sent in reverse direction to request change in these flags (and the PERIOD field). The transition among the modes is shown in Figure 28.

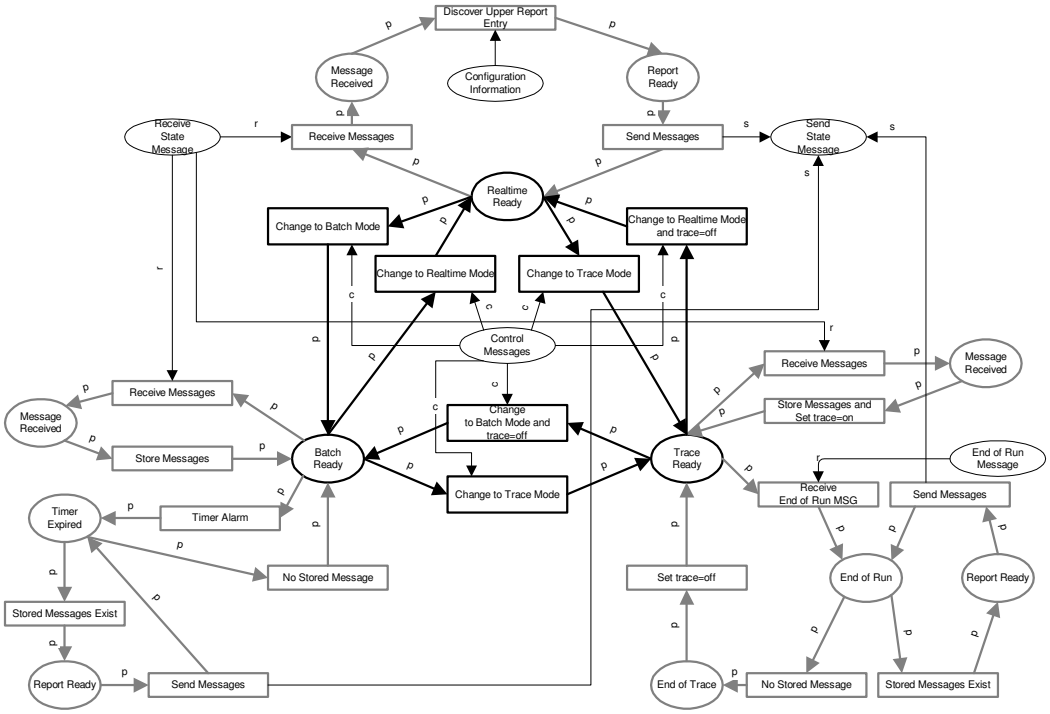

Figure 28 The Monitoring Point Status Transition Diagram

6.6 Visualization of Concurrent MTO Channel

The description of the concurrent MTO channel (CC) was given in section 5.4. In this section, the additional materials for visualization information of the CC are described.

A concurrent MTO channel (CC) is a simple but powerful MTO channel which is not available in current TCP/IP communication networks. The concurrent MTO channel (CC) is using a sub-MTO channel altered routing channel (ARC) to guarantee that the established MTO channel does not have an overlapped path from source to destination.

When the concurrent MTO channel system is running, the channel components generate status information. Because of the ARCs are used by the CC, the components in ARCs are also generate status information. This status information is delivered to the visualization system and displayed in proper format that was chosen by a channel monitor/user. The visualization system should support different views of similar messages based on the user's convenience. Each service component status transition is shown in Figure 29.

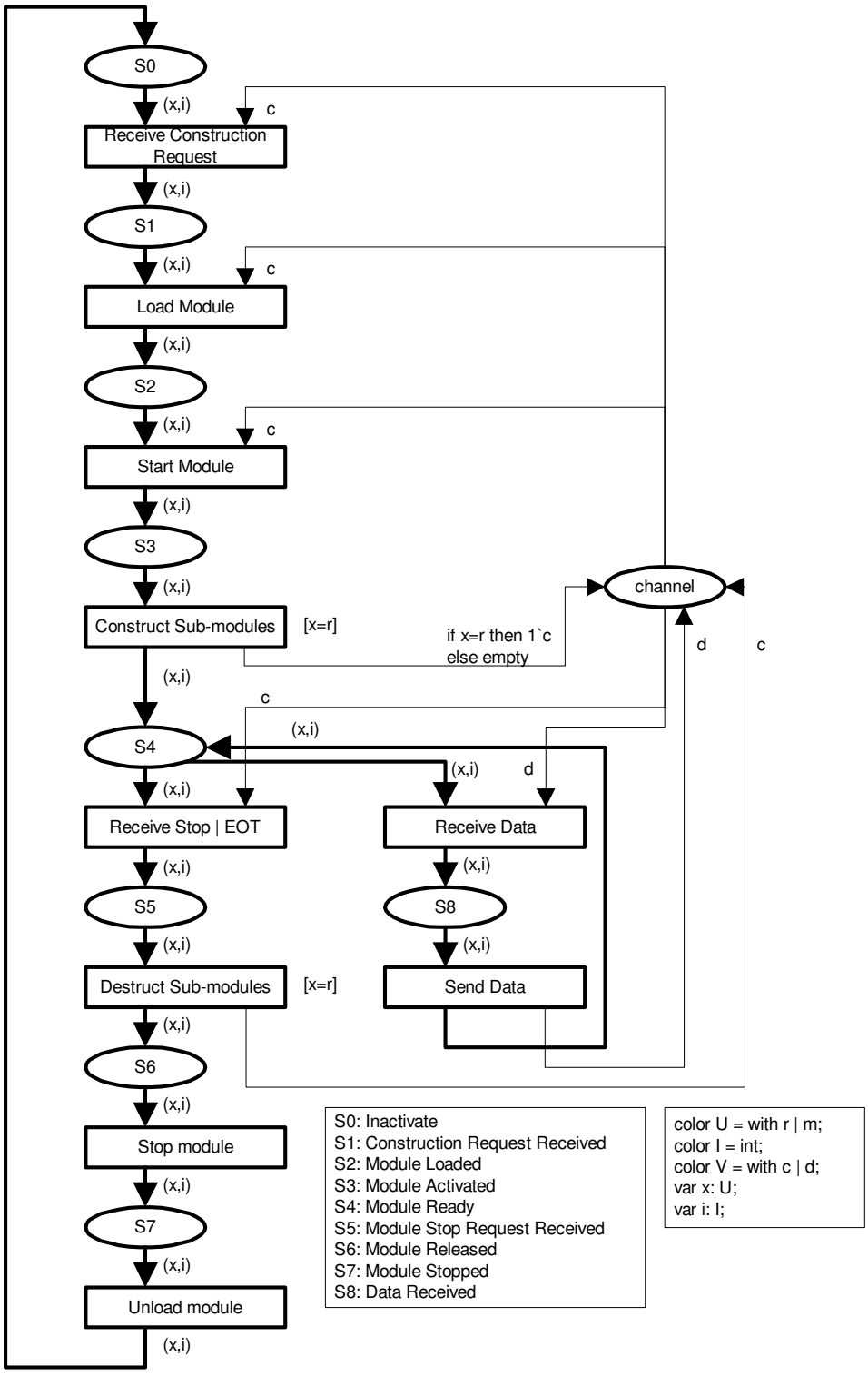

Figure 29 The MTO Channel Component Service Status Diagram

Figure 30 is a snapshot of MTO channel control center program for MTO channel control and status report. The MTO channel control center program is a central control program for a MTO channel construction system. It sends initial configurations to a MTO channel construction system and monitors each MTO channel control system node's status and events.

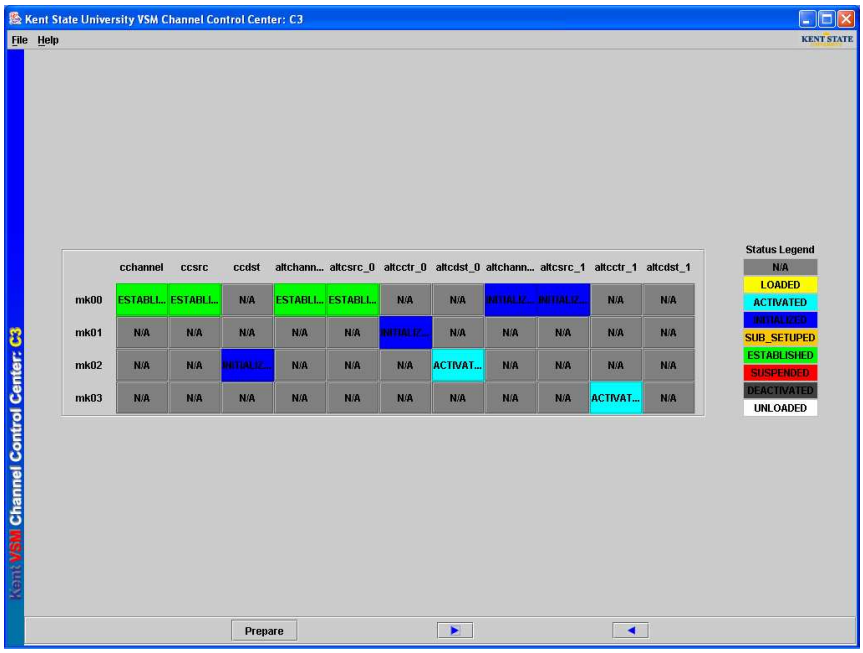

Figure 30 A Color Coded MTO Channel Control Center Control & Status Window

Hierarchical view of the same system status view is shown in Figure 31. The component colors represent their current status.

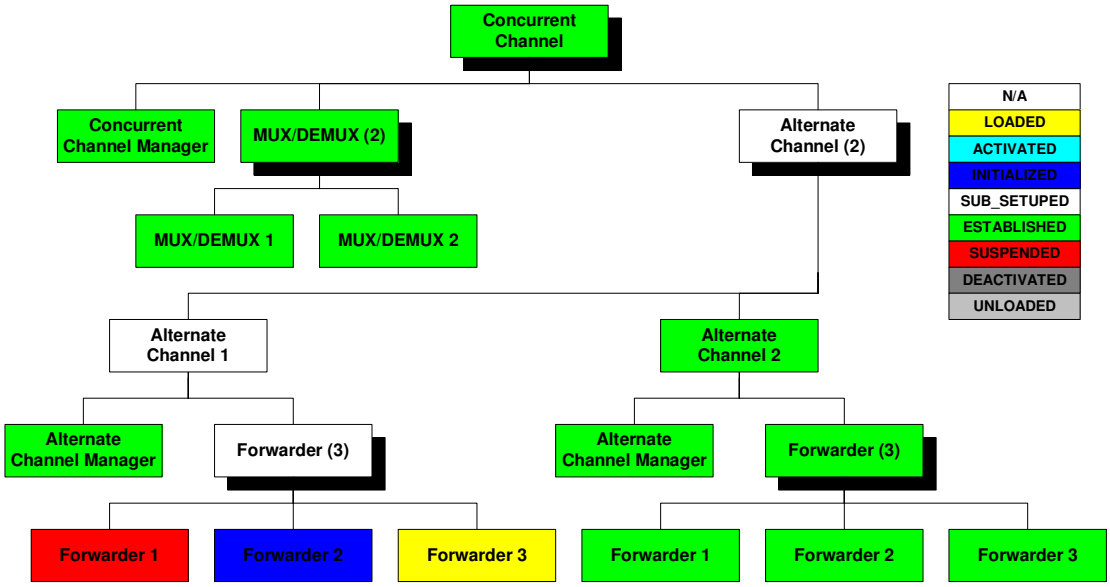

Figure 31 A MTO Channel System Hierarchical Process View

Each system component has its own status, for examples: loading, activation, initialization, established, unloaded, and so on. The status change information of the system is stored and replayed any time later for detailed investigation. The review interface shown in Figure 30 consists of simple forward and backward buttons but the value of the function cannot be ignored.

CHAPTER 7

An Additional Example of Service Transport MTO Channel

7.1 SONET MTO Channel

Adaptation is a fundamental phenomenon in natural systems. It seems that engineering of any large and complex system intrinsically requires the inbuilt ability of its components to adapt. Internet has already grown into a mega net with global reach. Now, with the emerging need of advanced applications, it is poised to evolve into a complex system of systems. With its expansion, the asymmetry of the Internet is also increasing. Historically, the initial Internet architecture had been conceived to cope with the heterogeneity of network standards. [40] Before the problem had been solved, it now appears that a second era is underway of in progress. It seems that the next generation of the Internet will have to deal with more intrinsic (and perhaps harder to overcome) heterogeneity— the asymmetry of hard network resources such as bandwidth, or switching capacity. [41] This asymmetry can evolve from the fundamental physical limitations at the fringe of extreme technology such as the power crunch in an intergalactic network element, or from something as mundane and insurmountable as socioeconomic disparity,- i.e. the digital divide.

In this section, a MTO channel that concentrates on creative adaptation is presented. There are quite a few works tending adaptive systems—particularly in the areas of scalable video communication, web caching, and very recently in mobile information

systems. This section presents a MPEG-2 rate transcoding MTO channel, which addresses the issue of adaptation from two levels. It adapts with respect to two critical network resources— bandwidth and the processing resource at the junction nodes. While the link bandwidth adaptation has been addressed up to some extent in a few of the recent research studies little attention has been paid to the node capacity adaptation. The transcoder senses local asymmetry in link capacities at various junction points of a network. Based on that, it accordingly adapts the video stream rate. On the second level, the transcoder MTO channel also senses the local computation power to execute its rate adaptation task. And thus, based on the network computational power, it demonstrates self-organization behavior. In each of these adaptive behaviors, it employs a number of techniques. For rate adaptation in the first stage it uses full re-quantization-based transcoding. For extreme rate scalability, it further employs a focal object based region discriminating encoding. To adjust with the processing power problem, it first can shift back to a low computation mode of transcoding using motion vector computation bypass. However, when a single node becomes insufficient, it dynamically migrates computations to neighboring nodes in search of increased processing power. The available bandwidth and computation resources are subject to change during the run time, and the measurement methods of the resources are can be enhanced. In this paper, however, shows one possible implementation for a video transcoding MTO channel that can deploy its components in network based on the available resources and can adapt computation and bandwidth change. The implemented MTO channel is called Self-Organizing

Network Embedded Transcoder (SONET). This section provides an architectural overview of this MTO channel system.

7.1.1 SONET MTO Channel Architecture

SONET is a full video transcoding MTO channel. It transcodes a video stream to adapt to the user's requirements, and the available computing and network resources. It deploys its components based on the computing and network resources. The MTO channel's components are SONET_mgr, SONET_enc, and SONET_mux. Depending on the component's needs, the SONET MTO channel deploys as many SONET_encs as possible to fulfill the user required transcoding rates. The MTO channel connection and its components are shown in Figure 32.

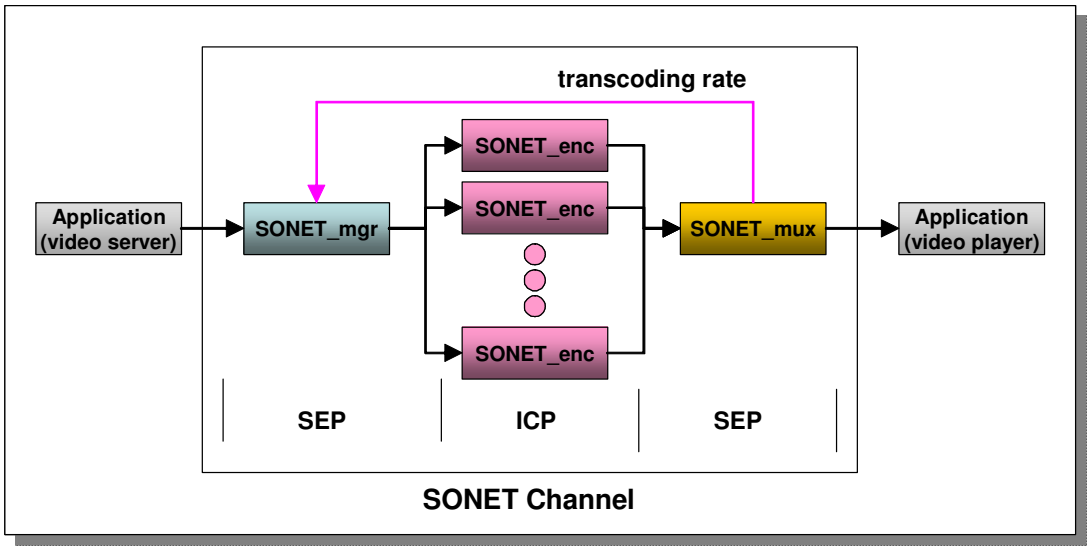

Figure 32 The SONET MTO Channel Architecture

SONET_mgr is a manager component and it will be located in a SEP. The SONET_mgr decodes the video stream and multiplexes the decoded video stream to SONET_enc. The video stream is processed in a video segment unit. The SONET_mgr includes a scheduler to distribute video segment units among available SONET_encs. The SONET_mgr also works as an event handler which is generated by SONET_mux for reporting transcoding rate information to adapt network and computing environment where the MTO channel runs.

SONET_enc is a video encoding component and located in ICP. In SONET MTO channel, SONET_enc is performing transcoding work as well as forwarding the transcoded contents to the SONET_mux. SONET_enc is receiving video segment units to transcode from SONET_mgr, which is a manager of the SONET MTO channel. Besides the video segment unit to transcode, SONET_enc also receives parameters from the SONET_mgr which are used to transform the input video segment unit to output video contents. The parameters are configured to adapt current network and computing environment which is of course coordinated by the MTO channel manager, SONET_mgr.

SONET_mux is a multiplexing component in a SONET MTO channel and is deployed in a SEP. It sends transcoded video segment units in sequence. It generates events which report current transcoding rates in each encoding path. The events are handled in the manager component, SONET_mgr. The events will trigger actions in SONET_mgr to schedule encoding paths, change number of encoders to use, and generates parameters for SONET_encs to adapt the run time environment change.

7.1.2 Visualization of SONET MTO Channel System

The application deployment map is shown in Figure 33.

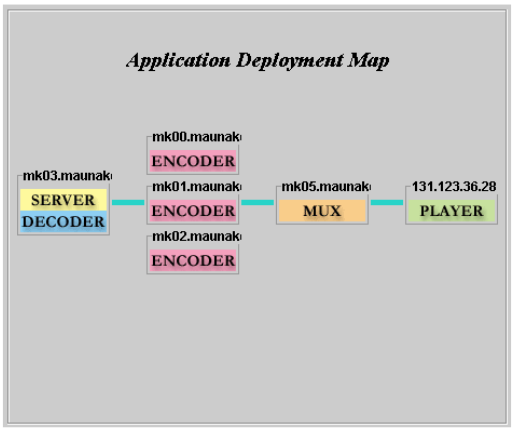

Figure 33 A SONET MTO Channel's Application Deployment Map

The SONET MTO channel's application deployment map shows where the components are located and how they are connected to each other.

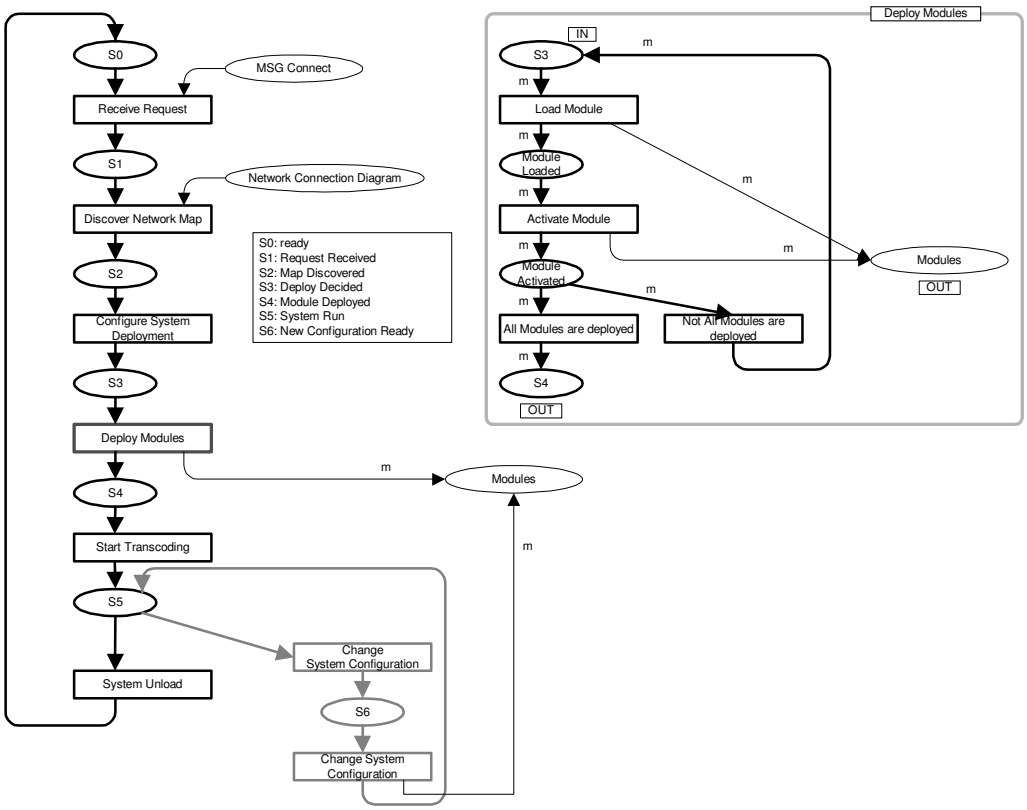

Figure 34 The SONET Service Status Transition

Figure 34 shows the whole system status window. The whole system status information is received by SONET's control center program as the system runs. It initiates service construction when it receives a SONET service request from an application. During its deployment phase, it uses a network connection diagram and generates a deployment map. The system deploys its components according to the deployment map. When the system is faced with environment changes, it reconfigures the deployment map and changes the system configuration during run time. Status messages are generated during the system's lifetime and the structure descriptions are loaded when the visual system is initiated. When the visual system receives a system status message,

the visual management system interprets the value with given system status message description.

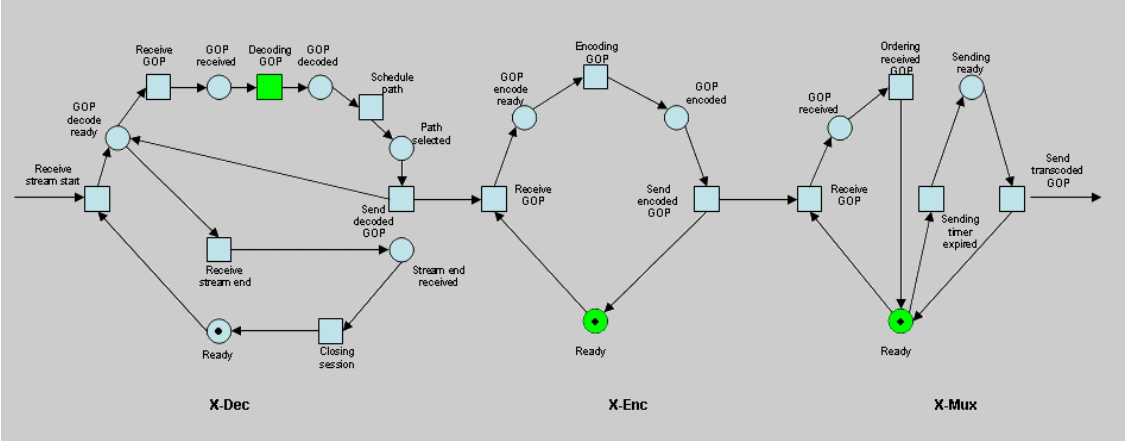

Figure 35 A Component Status Transition Diagram View of SONET System

The map displays computing systems with components on them and their connections. With component status descriptions, the status representation can be more intuitive as shown in Figure 35.

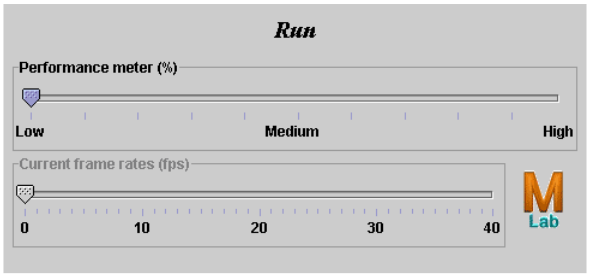

Figure 36 A System Wide Controls and Status Representation Window

The control and status management system should also support whole system controls and other system wide status. Figure 36 represents system wide control and status information. The performance meter is a control slide for the system performance. System performance value changes initiate the generation of control messages. The

generated control messages are delivered to the effected components. The performance of the system, in this case current frame rates, will have a new status as a result.

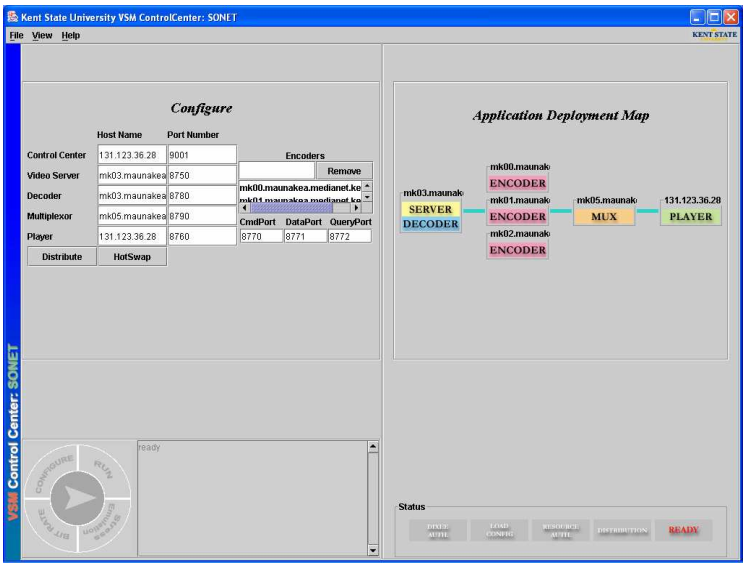

Figure 37 A System Configuration Window

Detailed status messages and control messages are available in the status window. Figure 37 shows the status window at the left bottom and the system configuration window at the middle left.

7.2 Jitter Controlled SONET MTO Channel

In the dynamic service composition MTO channel, a computation may not receive the same resource on all its runs or it may change even when a computation is underway. The computation can happen in chunks involving multiple locations. In between the data, pipes will transit streams of data from one computational unit to another. A key challenge in this vision of harnessing network technology for a giant computing MTO channel is

the complex and synchronized management of data streams between these computing entities.

The temporal characteristics of the information flow among the computational units make serious impacts on the time when the service end receives the processed information. This problem is quite different from its counterparts in classical networks. Delay-jitter management will be a central concern irrespective of the framework of networked computing. This will be a major and central concern for conventional time sensitive application processing such as media streaming. Interestingly, many other application processes, which are not normally known to be time sensitive, may become so. The new variability introduced by uncertain computation resources available over a loosely federated resource pool can seriously destabilize synchronization, load balancing, and the utilization efficiency of known distributed solutions.

A MTO channel development sometimes does not need to reuse an entire MTO channel; rather, it may customize an existing MTO channel component. The presented Jitter controlled SONET MTO channel is a video transcoding MTO channel as shown in section 7.1 with application-data-unit-level jitter and delay controls. Component reuse can be applied in this augmentation where it only needs to customize the scheduling component, SONET_mgr, and reuses the other components in the SONET MTO channel.

7.2.1 Related Works for Delay and Jitter Controls

The recent schemes proposed for jitter and delay control can be roughly categorized based on the traffic modeling (statistical vs. observation based adaptation), and the action level (end-to-end vs. network layer techniques). [27][28][29]Argiriou and Georgiadis suggested a technique for adapting the transmission rate of an application while maintaining the perceived quality at the receiver at acceptable levels. [30] When a new connection arrives in the system it renegotiates rates for all running applications. Khan, Yang, and Gu proposed the rate symbiosis technique between application and network transport, using an interactive generalization of TCP sending end-point. **[9**] It does not require any expensive end-to-end measurement. Zhang and Ferrari presented the rate control static priority (RCSP) scheme, which can provide multi-objective queuing including jitter optimization for Poisson like traffic distribution. [29] In this scheme, a component called regulator is assumed on each stream for traffic shaping based on its optimization objective. A component called scheduler was assumed on switches to resolve the priority for the multiplexed flow. Boorstyn, et al. has presented an algorithm for providing statistical assurance which they call “effective envelope” for traffic scheduling and demonstrated it for optimizing jitter at an intermediate node where multiple video connections between multiple sender and receivers intersect. [27] The method assumes continuous-time fluid-flow traffic. Each intermediate node has one input regulator for each stream, and a scheduler. The regulators and the scheduler work jointly.

Bennett, et al. used special features, called Expedited Forwarding[1] (EF), to guarantee delay jitter bound in Differentiated Services architecture. [31] The packet forwarding at a specific rate is guaranteed if (i) a connection is admitted at connection time in EF PHB, and if (ii) the traffic obeys the assumed idealized statistical distribution.

The observation-based controls monitor traffic in each node, rather than relying on any static traffic model. The monitored information is fed to a scheduler at switch. Rexford, et al. argue that knowledge of a traffic pattern is not easy to obtain, or is limited, as in the case of live video conferencing. [28] They show a Hopping-Window smoothing. Stone and Jeffay described a policy called queue monitoring, which observes delay jitter and dynamically adjusts display latency for low latency conference calls. [32] By monitoring display queue, it retrieves changing end-to-end delay and corrects jitters without time synchronization. Mansour and Patt-Shamir also suggest jitter control algorithms by monitoring buffer fill rates. [33]

Compared to the previous works, this paper addresses the problem with respect to joint communication and computation delay. This is unique to the Internet based computing environment. Dynamic path resource estimation is used in the suggested sytem. Information stream centric computing adds a number of new challenges. Even in a real network environment, it is difficult to obtain the source traffic model. In the active paradigm, network computation adds an additional set of complex variability. All network nodes do not have the same processing capability. The processing time can vary for different contents and for the degree of customization. The initial data can

[1] RFC 3246 defines the Expedited Forwarding Per-Hop Behavior (PHB) with the intent to provide a building block for low delay, low jitter and low loss service by ensuring that the EF aggregate is served at a certain configured rate.

dramatically alter in size and time spacing at each stage of servicing. The capsule data unit can be of unequal size. All packets are not uniformly needed by the service capsules. Also, there is an effect of non sequential access. Some of the packets should be used at the same time by the service component, while some others may not be accessed at all. In this paper, a joint buffering and scheduling based algorithm are demonstrated which corrects both computation and transmission difference to reduce the jitter variations to get jitter free play of a video stream.

7.2.2 Multi-path Jitter Model

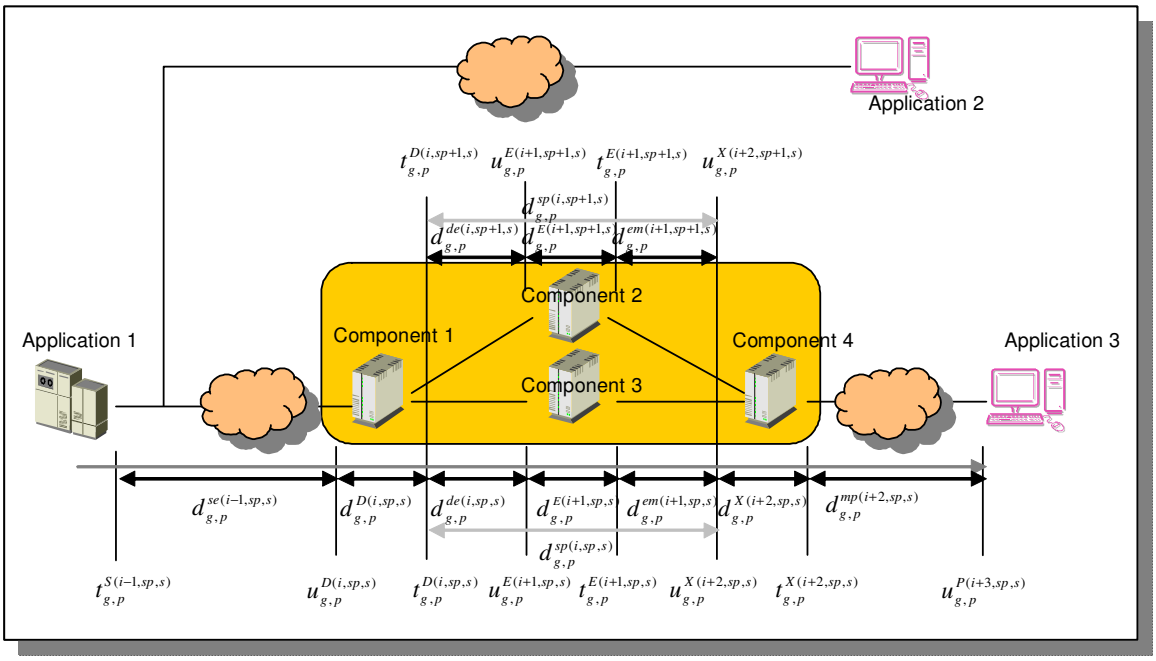

Figure 38 The Multi-path Jitter Model

In this section, a jitter analysis framework for a multi-path computation MTO channel is provided first.

In any computing service, the delay (and jitter) can occur not only in the network pathway during the transmission, but also in the processing capable nodes during their processing. Therefore, overall delay contains computation delays as well as transmission delays. The model will include both.

The notation will be explained first. The notation is rather complex because of the three-level mapping required between the flow, processing and the network. For denoting the delays, the following two-level notations are used. Subscripts are used to refer to the ADU's sequence number (g) and the path number (p). Each ADU is processed by a set of sub-task components (M). There can be multiple instances of a component in a network. First each sub-path should have a copy of each component. Also, for some type of services (such as tree-computing in a multicast distribution scenario shown in Figure 38) an information stream on its way can encounter multiple services with recurrence of the whole set of transformations. Components are also ordered and have a stage index. Therefore, in the superscript, each component's M is also identified with its *stage index* (i), *service number* (s) and the *sub-path* number (sp) within this service. These three appear as an argument of the stage name. Thus, let g, p, sp, m, s denotes respectively the g-th ADU, path number, sub-path number, component name, and the delay stages in a service. Then the delay experienced by an ADU along a path p can be expressed as:

$$D_{g,p} = \sum_{m}^{M} \left(d_{g,p}^{m(i,sp,s)} \right) \tag{1a}$$

For example the computing service is defined by a set of sub-task processing stages:

$$A \subseteq \{D, E, X\}$$

Where, *D, E, X* respectively represent the computation delays in the computing unit D, E, and X units. Their orders are 1, 2 and 3 respectively. Thus the total delay stages include the communication delays as well:

$$M \subseteq \{D, de, E, ex, X\}$$

Here, *de* and *ex* represent the communication delays in the first and second stages respectively.

Thus the objective of the proposed algorithm is to reduce the variation in inter-departure time from the joint defined by (1b):

$$J = \sum_{g}^{Stream} \left| D_{g,p} - D_{g-1,p} \right| \tag{1b}$$

Each component has a computation delay. This computation delay can be shown as in (2).

$$d_{g,p}^{a(i,sp,s)} = e^i \times r_c^i / B^i \tag{2}$$

Here, e^i is the input ADU size in bits, r^i_c is a computation needed for the component in flops per input bits. B_i is the processing power on the node running the component (or cycles allocated to the service) in units of flops. After the stream flows via a processing capable component, its size can change. The output stream size is represented by a *stage expansion factor*. The stream size after the i-th stage is thus expressed as in (3).

$$f^i = F \times \prod_{j=0}^{i} r^j \tag{3}$$

Let f^i is an output stream size while F is an initial input stream size and r^i is a *stage expansion factor* or output bits per input bits of a stage i. The relation of f^i, e^i and r^i is shown in (4)

$$r^i \times e^i = f^i \tag{4}$$

The delay in a link is shown in (5).

$$d_{g,p}^{ij(i,sp,s)} = f^i / B^{ij} \tag{5}$$

Here, f^i is an output stream size as seen in (3) while B^{ij} is a bandwidth of link i.

7.2.3 Algorithm

Following is the structure of the control algorithm. The logical components dynamically estimate the incoming link bandwidths and computation rates experienced by the ADUs. Following the same flow path the estimates then flow downstream into the joint component. The joint node then sends the aggregate feedback back to the fork point that dynamically schedules the newly arriving ADUs along the sub-paths to reduce the overall jitter and timely processing of the ADUs.

Proper sub-path selection for a given ADU in a stream is critical to reduce delay jitter in a stream having several processing stages. Selection of proper sub-path is based on the delays measured along the competing paths.

7.2.4 Scheduling

Given the streaming rate (R), the algorithm estimates a relative *target arrival time* (Tg) at the destination for each ADU. A quantity *maximum allowed delay* is estimated for each ADU based on this deadline.

The algorithm chooses a least weighted time path (to be explained shortly) among the paths which have predicted delay less than the maximum allowed delay.

When there are multiple conforming sub-paths, the weighted time is a time based on *the average delay time* and the *delay variation* of the sub-path. If no sub-path could process a given ADU within the deadline for it, then the least delay time sub-path is chosen without considering the variations.

The algorithm starts optimistically. At the start of the flow, the average delay is initialized to the lowest possible delay of the path and the delay variation is initialized to zero. During the run time, the average time and delay variations are adjusted by measurement on each sub-path. The *joint* gathers individual delays and delay variations from each sub-path and informs the values to the scheduler in *fork*. So, even if the initial values are not correct, the algorithm improves the estimates as processing progresses.

7.2.5 Delay Estimation

An expected sub-path delay is the sum of (i) expected delay of transmission from fork to the first sub-path processor, (ii) sub-path processing time, and (iii) transmission

time from last sub-path processor to the joint. The following equation is used for deriving expected delays along each sub-path:

$$\tilde{d}_{g,p}^{sp(i,sp,s)} = \tilde{d}_{g,p}^{de(i,sp,s)} + \tilde{d}_{g,p}^{E(i+1,sp,s)} + \tilde{d}_{g,p}^{ex(i+1,sp,s)} \quad (6)$$

Links Capacity Estimation: As seen in (5), $\tilde{d}_{g,p}^{de(i,sp,s)}$ and $\tilde{d}_{g,p}^{ex(i+1,sp,s)}$ can be predicted based on f^i and B^{ij}, but f^i and B^{ij} may vary for several reasons. The actual compression on each ADU can vary from the ideal compression ratio. The network activities on a link may cause different B^{ij} values from time to time. So, each of the nodes in a sub-flow including the joint estimates the average values based on previous measurements:

$$\tilde{d}_{g,p}^{ij(i,sp,s)} = \tilde{e}^i / \tilde{B}_{g,p}^{ij(i,sp,s)} \quad (7)$$

The bandwidth for each incoming link is approximated by each receiving node, including the fork node, using the method shown in (8).

$$\tilde{B}_{g,p}^{ij(i,sp,s)} = \frac{B_{g(k-2),p}^{ij(i,sp,s)} \times (k-1) + B_{g(k-1),p}^{ij(i,sp,s)}}{k} \quad (8)$$

Here, k is the number of ADUs which arrived at the receiver node or arrived at the joint using sub-path sp. The g(k) is a k-th ADU number which passed through the sub-path sp. The join estimates the quantity separately for each incoming flow. If the path has no history then the last known or initially known bandwidth is used. Please note that the right hand side quantities of the equation are observed bandwidth at the joint, not a prediction.

Processing Capacity Estimate: Similar to the transmission delay, the component delay can also be different from the ideal expected value. Thus, averages are estimated here as well.

$$\tilde{d}_{g,p}^{a(i,sp,s)} = \frac{d_{g(k-2),p}^{a(i,sp,s)} \times (k-1) + d_{g(k-1),p}^{a(i,sp,s)}}{k} \times e^{i} + Q_{g,p}^{a(i,sp,s)} \tag{9}$$

Equation (9) is for delay of a component *a* (a⊆A). It is derived from the *average delay per bit* observed on the previous ADU's on the sub-path s*p* and the current input ADU size, e^{i}. Also, in each component, it has a queuing delay $Q_{g,p}^{a(i,sp,s)}$. In computing service all the components do not operate in identical speed. Each processing component thus maintains an incoming queue of unprocessed ADUs. There is negligible queuing delay on the component D. It is relatively fast though compared with the encoding speed. The encoder's queuing delay is given to equation (10).

$$\begin{aligned}
&Q_{g,p}^{E(i,sp,s)} = \tilde{d}_{c,p}^{E(i,sp,s)} - (T_c - T_s^{E(i,sp,s)}) + \sum_{W \in \mathrm{W}} \tilde{d}_{w,p}^{E(i,sp,s)} \\
&\tilde{d}_{c,p}^{E(i,sp,s)} : \textit{expected delay of current encoding ADU, c, in encoder in sub - path sp} \\
&T_c = \textit{Current time} \\
&T_s^{E(i,sp,s)} = \textit{Start time of current encoding ADU, c, in encoder in sub - path sp} \\
&W = \textit{unstarted ADU scheduled in the node.}
\end{aligned} \tag{10}$$

The delay variations of sub-paths are used to select a proper sub-path for a given ADU. Its use provides the worst expected delay time in each path and thus can help in selecting reliable path. Equations (11) and (12) are used to track delay variation.

$$\overline{Avr}(d_{g,p}^{sp(i,sp,s)}) = \frac{Avr(d_{g(k),p}^{sp(i,sp,s)}) \times k + \tilde{d}_{g,p}^{sp(i,sp,s)}}{k+1} \tag{11}$$

$$\overline{Var}(d_{g,p}^{sp(i,sp,s)}) = \frac{Var(d_{g(k),p}^{sp(i,sp,s)}) \times k + \left| \overline{Avr}(d_{g,p}^{sp(i,sp,s)}) - \tilde{d}_{g,p}^{sp(i,sp,s)} \right|}{k+1} \tag{12}$$

7.2.6 Buffering

To absorb the jitter, the joint maintains an optimum jitter buffer. The queuing delay in joint, $d_{g,p}^{x(i,sp,s)}$, do as not need to be considered in selecting a sub-path because the joint absorbs it. The joint buffer also absorbs the jitter in the multi-path flow. The joint tries to send the ADUs at a smooth rate to the source. In the joint, the buffer size is occasionally dynamically estimated to provide the jitter absorption while keeping the delay at a minimum. The delay variation and maximum delay is used to get proper buffer size. The following equations are used to get buffer size at the joint. First, the maximum probable path delay is estimated in equation (13) for each path. Then, the worst path delay is estimated in equation (14). In a similar way, the estimation of the worst observed path delay variance is calculated using equations (15) & (16). The buffer size is dynamically adjusted to accommodate the maximum expected delay including the worst possible overshoot indicated by the delay variance.

$$Max(\tilde{d}_{g,p}^{sp(i,sp,s)}) = Max(\tilde{d}_{g,p}^{de(i,sp,s)}) + Max(\tilde{d}_{g,p}^{E(i+1,sp,s)}) + Max(\tilde{d}_{g,p}^{ex(i+1,sp,s)}) \quad (13)$$

$$Max(\tilde{d}_{g,p}^{s}) = Max(\tilde{d}_{g,p}^{0,s}, \tilde{d}_{g,p}^{1,s}, ..., \tilde{d}_{g,p}^{sp,s}) \quad (14)$$

$$Max(Var(\tilde{d}_{g,p}^{sp(i,sp,s)})) = Max(Var(\tilde{d}_{g,p}^{de(i,sp,s)})) + Max(Var(\tilde{d}_{g,p}^{E(i+1,sp,s)})) + Max(Var(\tilde{d}_{g,p}^{ex(i+1,sp,s)})) \quad (15)$$

$$Max(Var(\tilde{d}_{g,p}^{s})) = Max(Max(Var(\tilde{d}_{g,p}^{0,s})), Max(Var(\tilde{d}_{g,p}^{1,s})), ..., Max(Var(\tilde{d}_{g,p}^{sp,s}))) \quad (16)$$

$$BufferSize = (Max(\tilde{d}_{g,p}^{sp,s}) + \alpha.Max(Var(\tilde{d}_{g,p}^{sp,s}))) \times No.of\ ADU\ (per\ \text{sec}) \times Average \quad (17)$$

Where α=1 is typically used.

7.2.7 Scheduling Algorithm

A sub-path is selected for processing each ADU, a part of the information stream. The selection is based on the processing delay of the sub-path. First, it selects available sub-paths satisfying targeted delay of the ADU of the information stream. Among the available sub-paths satisfying the minimum delay constraints, it chooses the sub-path with the lowest variability. On the other hand, if there is no sub-path satisfying the targeted delay, then it chooses the lowest delay sub-path without considering the variability.

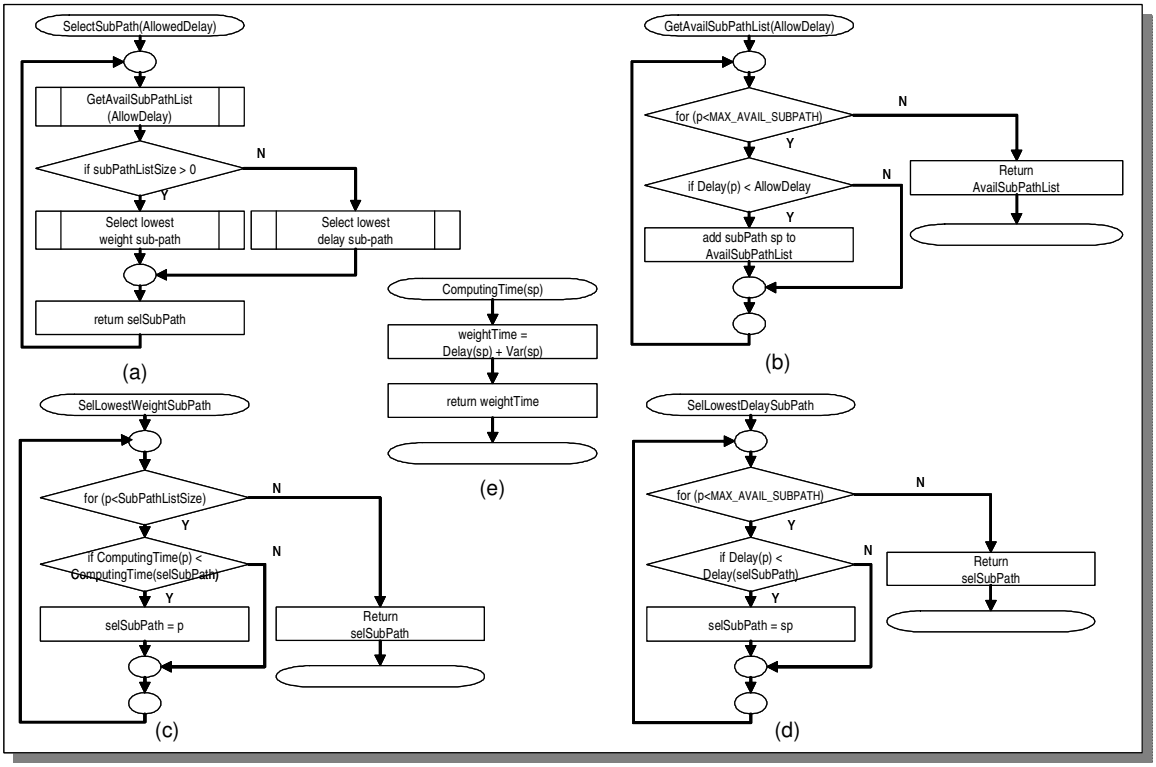

Figure 39 Sub-Path Selection Algorithms

7.2.8 Complexity of the Path Selection

The flowchart of the path selection algorithm is given in Figure 39. The time complexity of *GetAvailSubPathList(), SelLowestDelaySubPath(), and SelLowestWeightSubPath()* take *O(sp).* The time of *ComputingTime()* take *O(1).* The time of SelectSubPath() =*O(sp)+{O(sp)* or *O(sp)}*. Therefore, the *SelectSubPath()* takes *O(sp)* while *sp* is the number of sub-path. Generally the number of sub-paths is relatively small. So, the algorithm is reasonably fast.

CHAPTER 8

Performance Analysis

8.1 Test bed

8.1.1 ABONE

The suggested active MTO channel system (AMCS) does not need to have a specific hardware platform or software infrastructure. However, the experiments of the AMCS have been conducted on ABONE, launched under the DARPA ANI Initiative. [25] ABONE is an operational network and provides an Internet wide network of routing as well as processing capable nodes. Providers can contribute a confederation of computing capable nodes. Independent application involving multiple trust domains can be securely launched and executed. ABONE currently has about 100 nodes. The nodes are available from Europe, Asia and North America. DARPA's goal is to achieve about 1000 nodes. Resources in individual nodes are contributed and managed locally and independently by the contributing site administrators. However, the administrators do not have to manage the remote users. Authenticated remote applications can install and execute programmed components on any collection of these nodes via the ABONE backbone management and control backplane being a part of a centralized user pool. The codes are distributed via an enlisted set of *Trusted Code Servers* (TCS), which help authenticating them prior to distribution. The security domains are handled by the backplane control system. The backplane is being maintained by the ABONE Coordination Center (ABOCC) at ISI at

the University of Southern California. ABONE status can be monitored live from the ABOCC web site. [15] Technical information about ABONE is available from ABOCC. However, in this section a brief elaboration of its architecture and security relevant to our experiment is provided.

8.1.2 ABONE Software Architecture

The software structure of ABONE node involves a native Node Operating System (NOS) and Execution Environments (EEs). Current ABONE nodes are running on a variety of underlying NOS including Linux, Solaris and FreeBSD. Each EE acts like a remote shell providing a programming construct to a processing capable component. Nodes can support multiple EEs. Each EE can run multiple applications from multiple user domains concurrently. A number of EEs have been developed. Currently ABOCC permanently supports ASP, ANTS, and PLAN. [14][26] The management & control of ABONE nodes is provided by the ANETD system. [15] The ANETD allows users to obtain secured and controlled access to the ABONE resources. Its central function is to provide a safe execution environment for programmable components from multiple trust domains under the supported EEs. It itself has been designed as a special EE (account ABOCC) to support its own management, such as starting, stopping, monitoring and upgrading of the ANETD. At the core of ANETD is a robust access control and security model that enables users, codes, and nodes to be authenticated without individual node administrators to worry about them.

8.1.3 ABONE Security & Authentication Model

ANETD uses 512 bit public key cryptography to authenticate control commands. Each control command sent to an ANETD is digitally signed to ensure that ANETD access control policies are soundly enforced. ANETD enforces access control by enforcing two overlapping security domains. (a) It allows execution, deployment and control commands only originating from a set of known pairs, which is maintained by a list called the access control list (ACL). (b) It downloads and executes code only from a set of trusted servers (HTTP servers or local files/directories) specified in trusted code-server list (TCL). There are two categories of ACL and TCL: master and local. If ANETD is running as ABONE node, than it first reads all master ACL/TCL files, which are maintained and fetched from an ABOCC server, followed by all available local ACL/TCL files, which are maintained by local ABONE node administrators. If a node is configured as standalone, ANETD applies all local ACL/TCL files. The access control list (ACL) contains client node list. Each list has client ID, public key of the client which is a 512-bit public key, and optional parameters. The trusted code-server list contains a list of servers represented by a URL form, which specifies a single file or the root directory of permissible download codes.

8.1.4 Concurrent MTO Channel Experimental Environment

In the test bed environment, Linux boxes in a gigabit Ethernet connection are used. A concurrent MTO channel and altered routing channels are used for performance

measurement of the MTO channel construction. Figure 40 shows the test bed. The network nodes run RedHat 9.0 and uses AMD Athlon XP 1800+ or 1700+ with 256MB Memory.

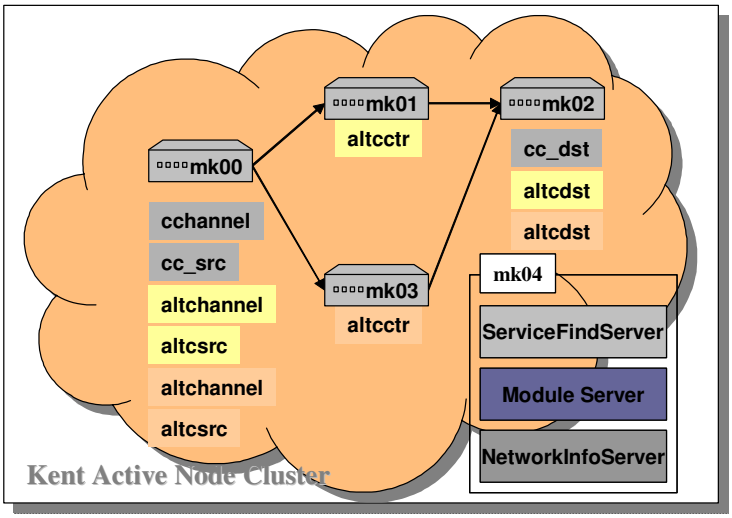

Figure 40 The MTO Channel Construction Test Bed

Table 29 shows the tested MTO channel's types and sizes. The concurrent MTO channel and the altered routing MTO channel are written in the Java programming language. They are compiled and executed in JDK 1.4.2.

Table 29 The Tested MTO Channel's Types and Sizes

Channel	Component Name	Component Type	Component Size (byte)
Concurrent MTO Channel	CC_mgr	Channel Manager	24531
	CC_src	Service Server	20133
	CC_dst	Service Server	18049
Altered Routing MTO Channel	ALTC_mgr	Channel Manager	16461
	ALTC_src	Service Server	14898
	ALTC_ctr	Service Server	14896
	ALTC_dst	Service Server	14890

8.1.5 SONET MTO Channel Experimental Environment

In SONET test environment, a total of five ABONE nodes are used, each machine runs RedHat Linux 7.1. They include three AMD Athlon 1.4GHz machines, one AMD Athlon XP 1700+ machine, and one dual Pentium III 450Mhz machine. Three encoders were used in the simulation. The nodes receive authenticated transcoding service components from a code server located at KSU Medianet Lab. Those are run on Athlon 1.4GHz, Athlon XP 1700+, and dual Pentium III 450MHz machines. The deployment, management and monitoring process was automatic and adaptive.

Selected nodes had different computation powers to make sure that paths have different delay variations in transcoding a video stream. The selected source video streams had identical contents but were initially encoded with different frame and ADU size (GOP size). The node and link capacities also had dynamic variations. There were other activities on the processing capable nodes as they were running on open ABONE.

The SONET MTO channel manager had a full graphical interface running on a local machine, and it provided the one point graphical visualization to the run time state of the entire component system. In the SONET MTO channel system, the system monitoring is built-in and thus the statistic information is gathered without any other modification. The status monitoring costs are already included in the signaling costs.

Table 30 The SONET MTO Channel's Types and Sizes

Channel	Component Name	Component Type	Component Size (byte)
SONET MTO Channel	SONET_mgr	Channel Manager	93127
	SONET_enc	Service Server	87881
	SONET_dec	Service Server	25351

Table 30 shows the SONET MTO channel's components, types, and sizes. The SONET MTO channel is written in the C programming language and compiled by GCC 2.96.

8.2 Experiment Results

8.2.1 MTO Channel Deployment Overhead

8.2.1.1 MTO Channel Optimization

The optimization is done before actual MTO channel deployment. The optimization is done when the channel deployment planning time. Figure 41 shows number of used channel components for GC(a,e) in section 4.5.

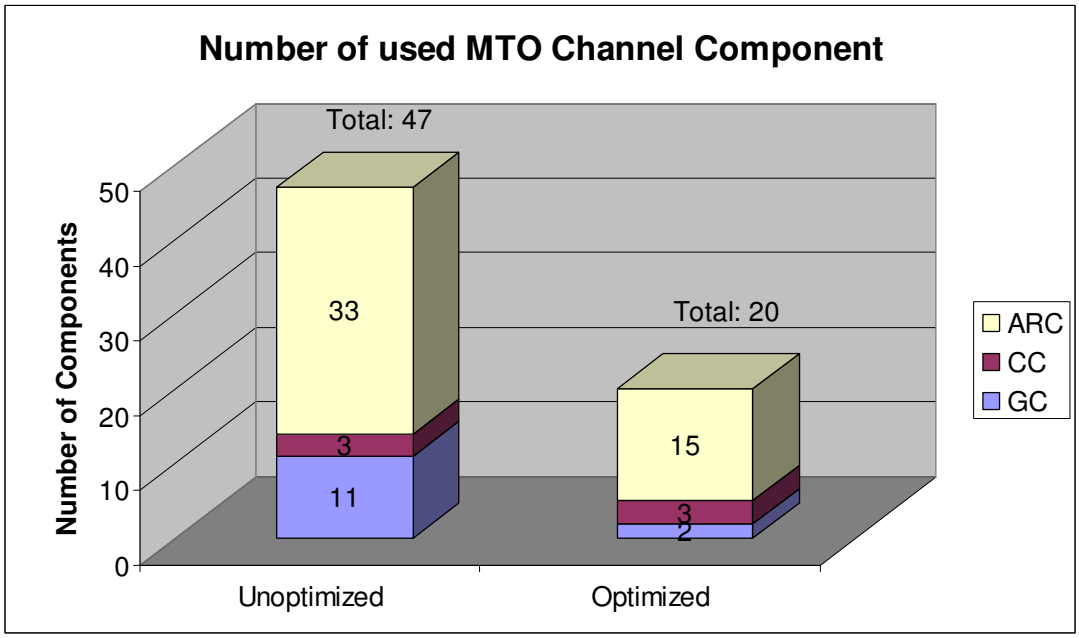

Figure 41 The Number of Used MTO Channel Component in GC(a,e)

GC has one component in the MTO channel, and CC has three channel components to deploy. The number of deploying channel components for the ARC MTO channel is three plus the number of intermediate computing points (ICPs) for ARC_ctr components. The number of MTO channel components for GC(a,e) is 47 (unoptimized) and 20 (optimized).

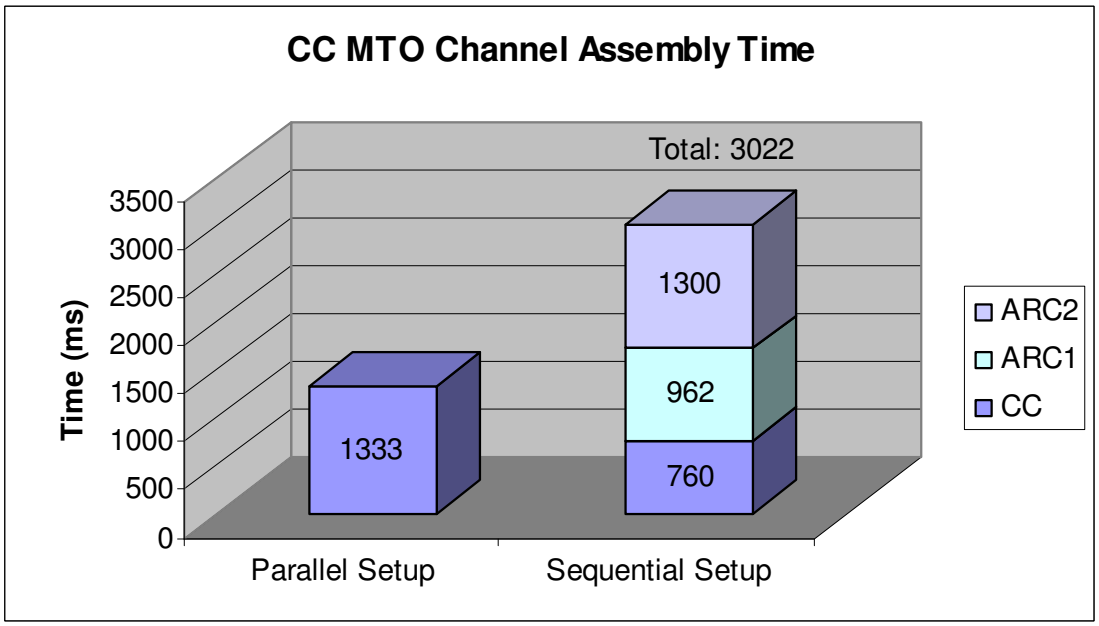

Figure 42 A CC MTO Channel Assembly Time

Figure 42 shows the MTO channel assembly complexity. If a MTO channel were constructed sequentially, it would take 3022 ms. However, when the MTO channel is constructed in parallel, as it normally is, it only take 1333 ms.

8.2.1.2 Concurrent MTO Channel

The following figures show the measurement result of MTO channel construction overhead.

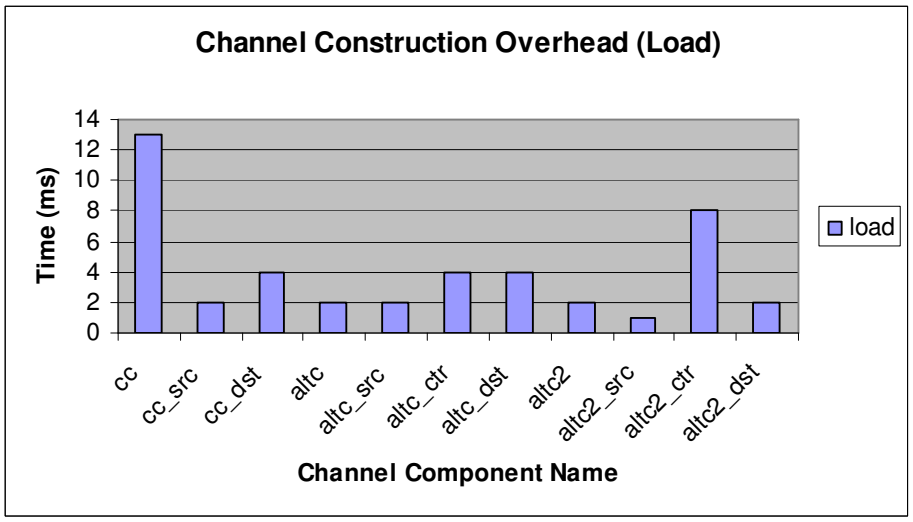

Figure 43 A Loading Overhead for CC MTO Channel Construction

Figure 43 is a graph for the loading overhead of the concurrent MTO channel construction. The differences of each component size mainly cause the loading time differences. The loading time is relatively small compared to the service time of a MTO channel.

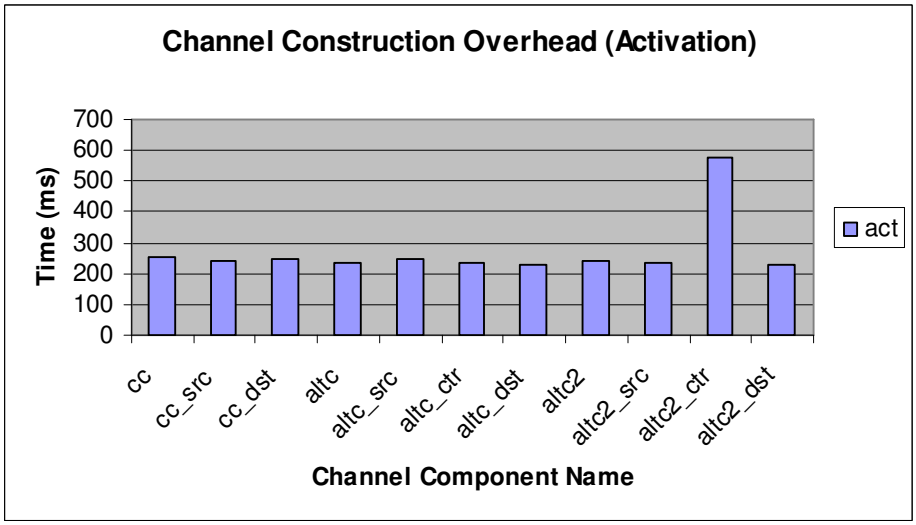

Figure 44 An Activation Overhead for CC MTO Channel Construction

Figure 44 shows the activation overhead of the MTO channel construction system. An activation time is a time period needed to activate a MTO channel component from the time of a specific MTO channel component invocation request to the time of notification reception from the invoked component. The time overheads are relatively larger than the other overheads because of the efficiency of the Java virtual machine. The MTO channel construction system was programmed in java language for its portability.

8.2.1.3 SONET MTO Channel

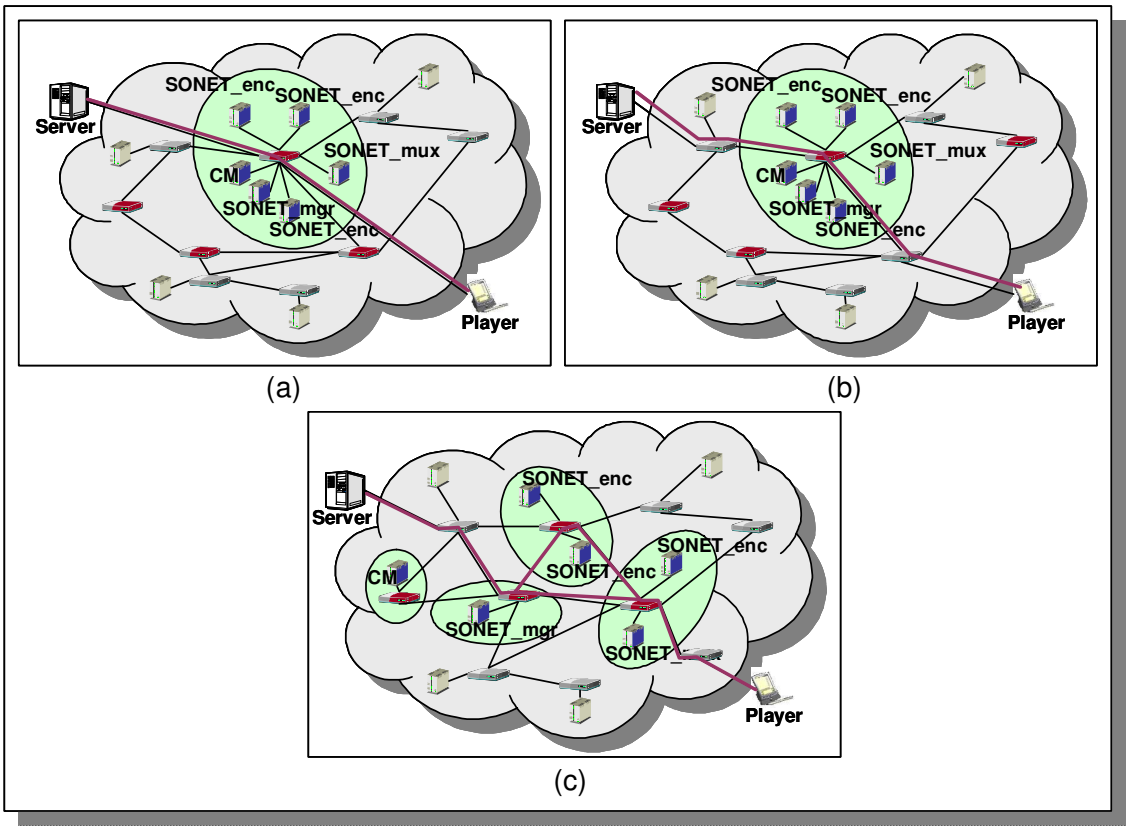

Figure 45 SONET MTO Channel Test Network Configurations

Figure 45 shows the network test bed for performance measurements. The system was tested in three test bed scenarios. In the first scenario, (shown in Figure 45(a)) the application as well as the SONET MTO channel components-- all were deployed in a single autonomous system's LAN. In the second scenario the application end points (server and players) were in different networks but SONET MTO channel computation was performed in a single network (shown in Figure 45(b)).

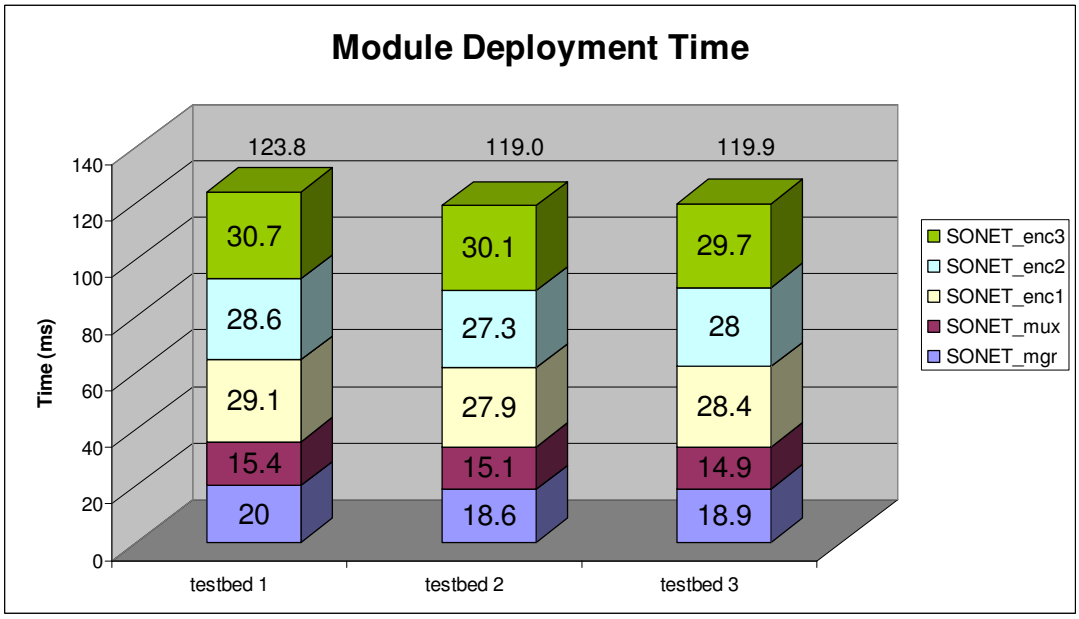

Figure 46 A SONET MTO Channel Component Deployment Time

In the third setup, the application as well as each SONET MTO channel component, was in distinct networks (shown in Figure 45(c)). The corresponding component deployment time of each test bed is shown in Figure 46. The component deployment time is averaged over 10 trials in each test bed. As shown in this figure, the first test bed takes more while the second and third test beds take approximately the same amount of time.

The stacks show how much time is taken by individual components of the system. In all three scenarios the total component deployment took about 1.2 seconds. It includes authentication, automatic component transfer and their activation in each processing capable node.

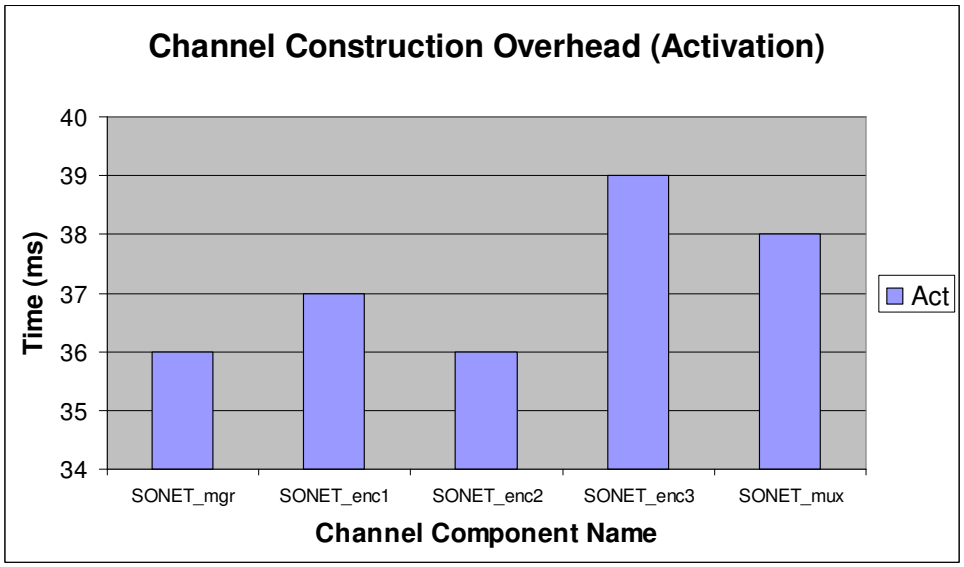

Figure 47 An Activation Overhead for SONET MTO Channel Construction

Figure 47 shows an activation time for SONET MTO channel. The SONET MTO channel is written in the C programming language. The activation time is faster than the time of concurrent channel, which is written in the Java programming language.

8.2.2 Signaling Overhead

8.2.2.1 Concurrent MTO Channel

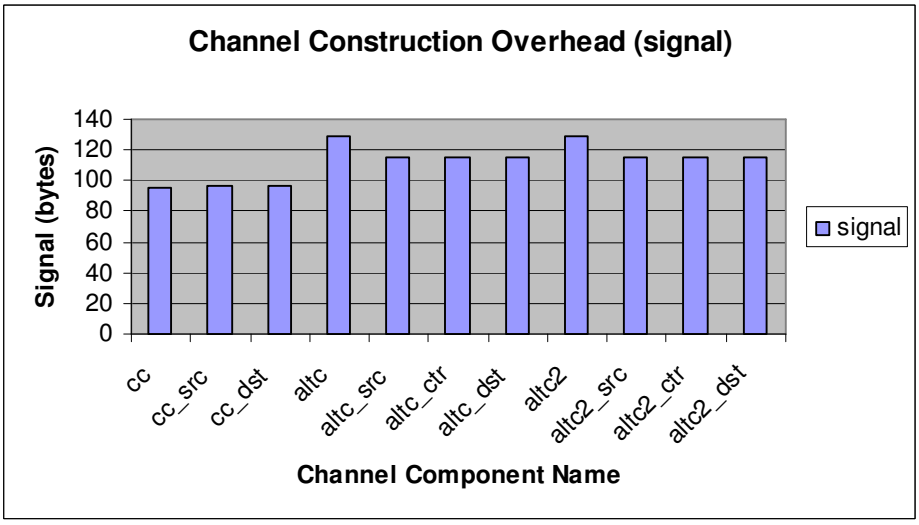

Figure 48 A Signaling Overhead for MTO Channel Construction

The signaling overhead for MTO channel construction is shown in Figure 48. The required signal is measured in byte size. The size of the signal is very small considering the size of actual communication data quantity.

8.2.2.2 SONET MTO Channel

The entire synchronization was performed by inter components signals. Clearly, a concern was how much communication resource was consumed by this. Therefore, the signaling overhead is logged in each component. The signal overhead is plotted in Figure

49. It plots individual SONET MTO channel components in scenario. For comparison, the second bar also shows the actual ADU data volume. This is a log plot. As shown in Figure 49, relatively small network resources are used for achieving coordination and control among the components.

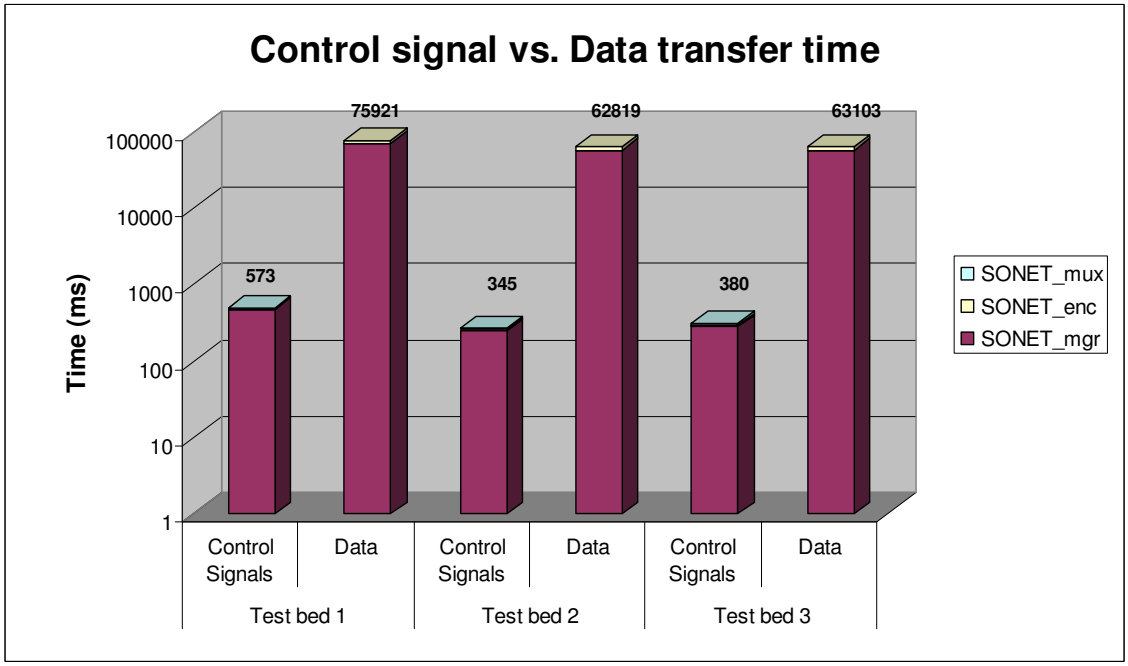

Figure 49 Control Signals vs. Data Transfer Time on Different Networks

It uses the same video. Thus, the transcoding time and control signals remain the same. Test bed 1 takes approximately 12 seconds more on transferring data than test bed 2 or 3. This indicates network resources are bottleneck in test bed.

8.2.3 Concurrent Channel's Transmission Speed

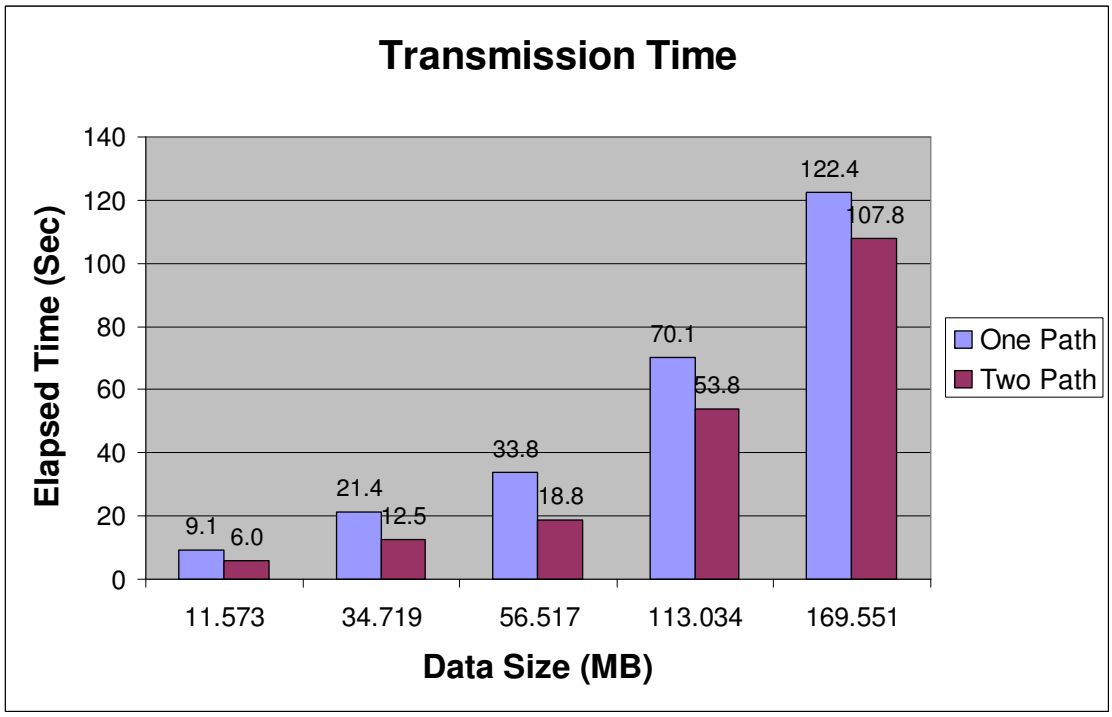

Figure 50 A Concurrent Channel's Data Transmission Time

A concurrent channel's data transmission time is shown in Figure 50. The experimented concurrent channel is used one and two transmission paths. The concurrent channel has performance benefit over single path transmission. Over the all data transmission size, the double path channel has advantage over single path channel. However, it receives less promotion when the transmission data size is small to fill its buffer than more data ready. The double path concurrent channel also consumes significant time on handling transmission data when the data is larger than it normally can process.

8.2.4 SONET MTO Channel's Cold Start Adaptation for Video Sizes

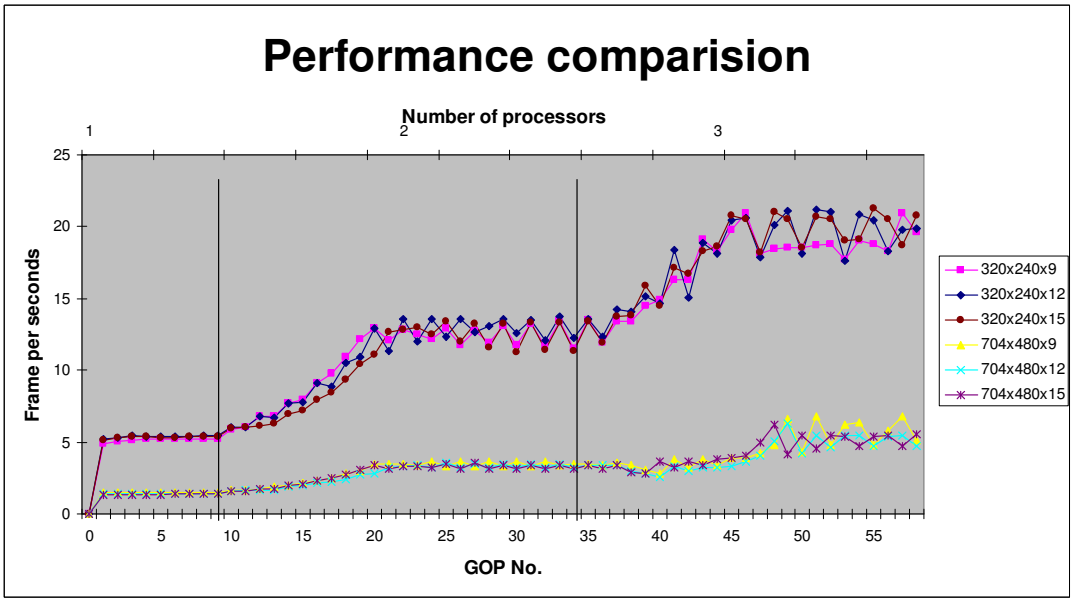

Figure 51 A Performance Comparisons among Different Frame Size and Computation

Figure 51 shows the frame-rate observed in their sample run on a small uncontrolled (with background computational and communication load) processing capable network consisting of 5 processing capable routers (with capacity ranging from 400 MHz ~ 1.5 GHz P4 processors, and the interconnections were 10/100 Ethernets with uncontrolled cross traffic). The system is deployed by itself and finds optimum mapping. Figure 51 plots the frame/ second statistics recorded at the GOP-MUX unit. It plots the performance for both 320x240 and 704x480 frame sizes streams at three different ADU (GOP) sizes. The computation load heavily depends on the number of macro-blocks and frame size. Based on the frame size the frame transcoding rate varied from 5-30 frames/second.

The adaptive behavior is noticeable at the step-like increments at the very beginning. Initially, the MTO channel used only one processing capable node. The single node was unable to sustain the target rate. Soon, it auto-deploys additional nodes. For example, for 704x480 video the second and the third nodes were deployed some time before the 20th and 60th seconds respectively. These delays represent the full feedback and effectuation delays. They include (i) the time to detect insufficiency, (ii) the time for stream auto deployment, and (iii) the time it takes the new results to appear at the MUX. As evident from the jumps, only three processing capable paths were available. This is dependent on the underlying network configuration.

The above results have been obtained from a transcoder running in full motion computation (FMC) mode [17]. While a general purpose transcoder supporting arbitrary video processing will require full encoding and decoding (used here), more special purpose processing can be performed by domain specific optimized computation. For example, further acceleration is achievable if motion vector computation bypass (MCB) mode is selected. However, the actual speedup is quite complex by the very nature of the paradigm. It will depend on the cost of full motion search which is also configurable, and the ability of computational paths (not only the computing power but also the required bandwidth).

8.2.5 SONET MTO Channel's Adaptation for Compute Power

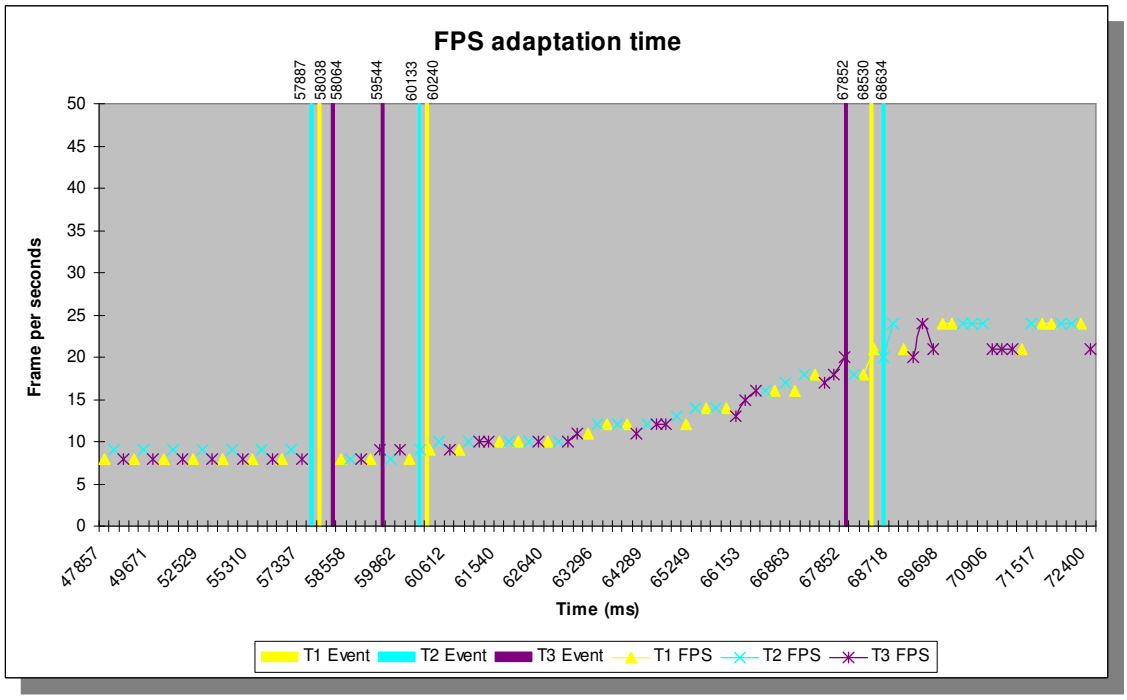

Figure 52 A FPS Adaptation Reaction Time

The SONET MTO channel system offers two forms of adaptation. The first is the change in the availability of the processing-capable nodes and the other is the change in their computing powers. In this experiment, incremental allocation of additional CPU power is emulated in the processing-capable nodes into the system in three steps (events T1, T2 and T3) by changing the *target frame rate* of the SONET MTO channel system. The corresponding change in SONET_mux buffers *throughput frame rate* (FPS) observed in the SONET_mux unit is shown in Figure 52. As shown in this figure, the reaction starts 1.5 to 2.2 seconds while the completion of reaction takes 9.8 to 10.7 seconds. Whenever there is a change in the network condition/ capacity the adaptive

system responds. It also shows the reaction time. More computation power gave more performance boost as expected. However, the first impacts of the events on the throughput were reflected in about 1.5 to 2.2 seconds. It took a little more time before the full effects took place.

8.2.6 SONET MTO Channel's Adaptation for Output Data Rate Change

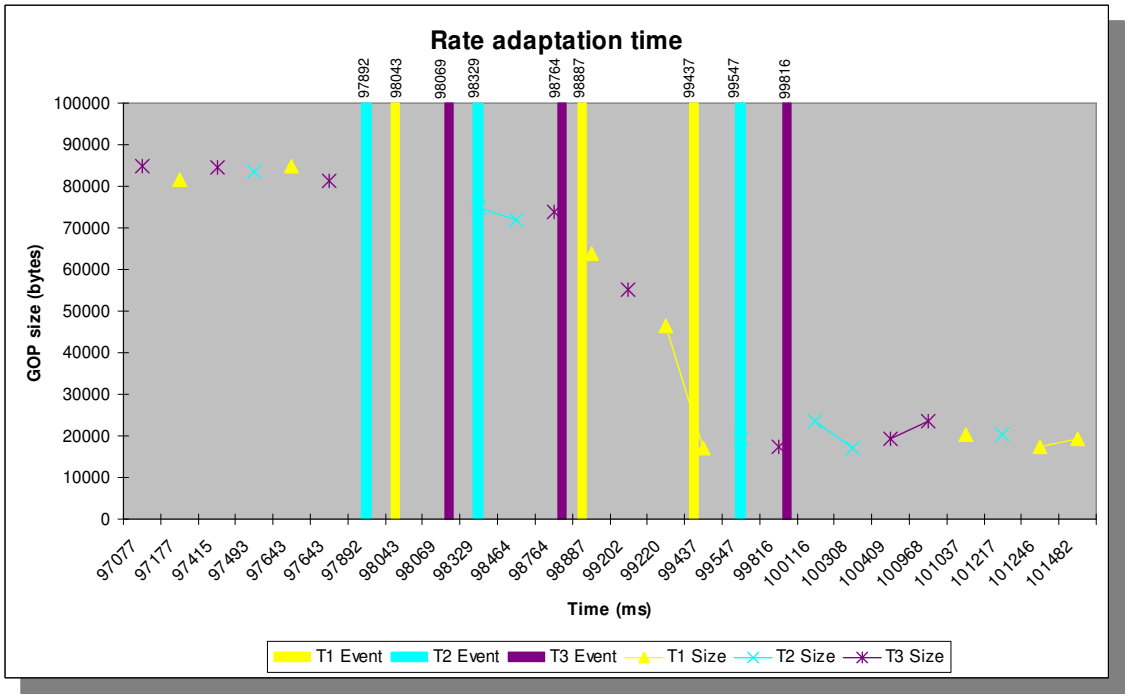

Figure 53 A Rate Adaptation Reaction Time

The other adaptation ability of the system is to adapt with respect to the change in communication capacity. The adaptability is emulated by changing SONET_mux's outgoing link bandwidth allocation from the transcoding system. This change automatically triggered the increase in the *transcoding ratio* of the SONET MTO channel system. In Figure 53, the corresponding rate adaptation trigger points and the reaction

time is shown. As shown in this figure, the reaction starts 0.4 to 0.8 seconds while the completion of the reaction takes 1.4 to 1.7 seconds.

Compared to the FPS adaptation, the rate adaptation's reaction time is faster, because SONET_enc can generate a rate adapted data output immediately in the middle of the transcoding, while the FPS adaptation is only shown after a whole ADU is carried and transcoded throughout the SONET MTO channel system.

8.2.7 Jitter Controlled SONET MTO Channel's Jitter & Delay Reduction

Figure 54 and Figure 55 show the simulation results. Figure 54 plots the jitter performance both with and without applying the technique. It shows the result for frame size 320x240. The x-axis is the GOP number (ADU number) in the video stream and the y-axis is delay jitter in seconds. Figure 55 shows the result of frame size 704x480.

As seen in Figure 54 and Figure 55, delay jitters are reduced dramatically with a delay jitter control scheduling. The first few ADUs have more delay jitter variations because the scheduler doesn't know proper initial delays of each path. After some time, however, the scheduler adapts in a running environment.

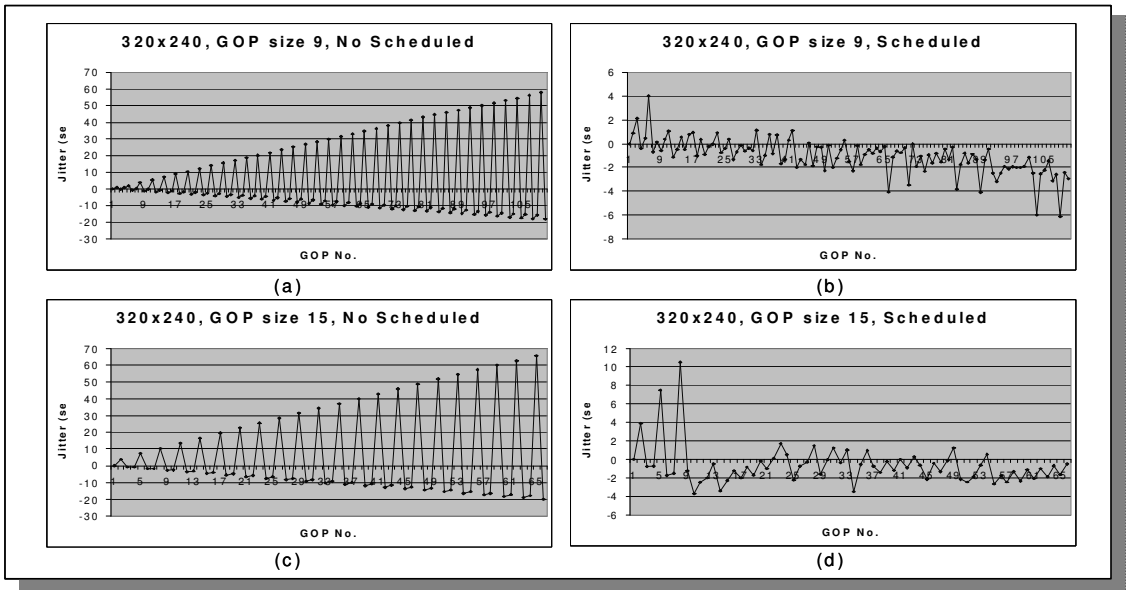

Figure 54 A 320x240 Video Stream Jitter Measurement

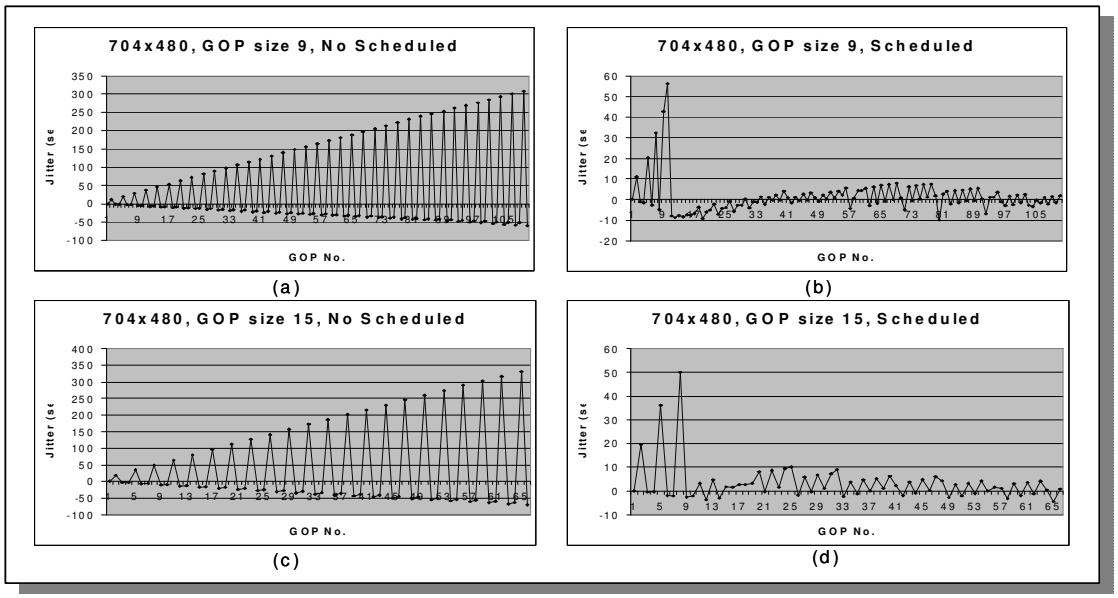

Figure 55 A 704x480 Video Stream Jitter Measurement

A larger-sized ADU video stream has more delay jitter variations than a smaller-sized ADU. This is due to the transcoding method. The encoders start encoding after all needed decoded video data arrives. Hence, the larger ADU size requires more time spent

waiting. Also, a bigger ADU stream needs more transcoding time than a small-sized ADU video stream. It increases the delay jitter variation. So, a larger-sized ADU video stream has larger delay jitter variations. If the encoder can start before all needed video data is transferred to them, it will reduce delay jitter more. Also, the results show that the larger frame video stream creates similar larger delay jitter variations. This is also for the same reasons – bigger frames need more transfer time and more transcoding time.

CHAPTER 9

Conclusion

Many anticipated advanced applications request transport speed that current network infrastructure can handle. However, until now, to support applications' requirements, each application has developed its own network control components at the ends of the communication entities. This covers some requirements, but has limited solutions. The programmable network extends the range and the power of network control by allowing programmability inside the pathway, as opposed to only the end-to-end control. However, current research has yet to provide any framework that supports systematical netcentric system composition formalism that offers language and platform independent code usability and the development scalability that follows from it. The suggested framework for a complex composition of a netcentric system is one of the first proposals towards this goal. The advantages of developed framework are shown below.

Layered service: Currently developed active network architecture is a flat layer architecture. All the active network connection is work independently from other active connection. An active network connection cannot use other active network connections to enhance its programmability on its traffic. Each connection is separate and could not work cooperatively. The suggested framework support uniform way to use other MTO channel to enhance or add the channels feature by cooperating with its sub-channels which are independently developed from the upper-level channel. The introduced

recursive channel construction hierarchical channel construction by recursively constructing its sub-channels. With the recursive channel construction, the framework maintains simplicity of channel construction but powerful enough to create complex channel construction.

Application service development: The framework includes the library which supports similar interfaces to current socket library that makes the transition from a conventional network program to a new MTO channel network program easier. Further the framework is giving uniform interfaces to a channel designer to develop their custom processing service by using other MTO channels which was developed by other channel designer. The framework opens a way to a 3rd party channel designer to develop more sophisticated and extensible channel to market, and application developers and other channel designers can more concentrate on their service by using the developed channel.

Limitation of the framework: The framework is created on the assumption that communication link for the channel control signal is available. The framework use lower level communication link functions to send and receive control signals and loading/unloading channel components during its channel construction and its service time. If the communication link is not available, a MTO channel could not deploy its channel components in network and could not construct the custom communication service.

The framework for netcentric MTO channel system requires little overhead yet is powerful enough to provide a construction for complex custom MTO channels. Using the recursive netcentric system construction methods, a MTO channel construction remains

simple and consistent, and the application can use its custom MTO channel as easily as the current socket library. The framework also provides a way of an 3rd party channel developer develops a channel and other channel developers use the channel to create their own channels like the development of current software object components. To be used in practical system, it needs to have fundamental channel service implementations such as a secure channel, a store-and-forwarding channel, and an error resilience channel. MTO channel depository and deployment architecture will leverage the MTO channel framework to more practical system to the public.

This work has been funded by the DARPA Research Grant F30602-99-1-0515 under its active network initiative.

References

[1] Seung S. Yang and Javed I. Khan, "Recursive Channel Construction in Network Centric Internet Computing System," US-Korea Conference on Science, Technology and Entrepreneurship, August 2004, Research Triangle Park, North Carolina.

[2] Seung S. Yang and Javed I. Khan, "Open Standard based Visualization of Complex Internet Computing Systems," International Conference on Computer Graphics, Imaging, and Visualization, Penang, Malaysia, July 2004

[3] Seung S. Yang and Javed I. Khan, "Open Standard based Visualization of Complex Internet Computing Systems," Workshop on Interactive Visualization and Interaction Technologies, Krakow, Poland, June 2004

[4] Javed I. Khan and Seung S. Yang, "Delay and Jitter Minimization in High Performance Internet computing," International Conference on High Performance Computing, Hyderabad, India, December 2003

[5] Seung S. Yang and Javed I. Khan, "Delay and Jitter Minimization in Active Diffusion Computing," IEEE International Symposium on Applications and the Internet, pp. 292-300, Orlando, Florida, January 2003

[6] Javed I. Khan, Patric Mail, and Seung S. Yang, "Flow Assignment in A Self-Organizing Video Stream that Auto Morphs Itself while in Transit via a Quasi-Active Network," 5th IEEE International Conference on High Speed Networking and Multimedia Communications, HSNMC2002, Jeju, Korea, July 2002

[7] Javed I. Khan, and Seung S. Yang, A Framework for Building Complex Netcentric Systems on Active Network, Proceedings of the DARPA Active Networks Conference and Exposition, DANCE 2002, May 21-24, 2002, IEEE Computer Society Press, San Jose, CA.

[8] Javed I. Khan, Seung S. Yang, Darsan Patel, et al., Resource Adaptive Netcentric System on Acitve Network: A Self-Organizing Video Stream that Auto Morphs Itself while in Transit via a Quasi-Active Network, Proceedings of the DARPA Active Networks Conference and Exposition, DANCE 2002, May 21-24, 2002, IEEE Computer Society Press, San Jose, CA.

[9] Javed I. Khan, Seung S. Yang, at el., "Resource Adaptive Netcentric System: A case study with SONET – a Self-Organizing Network Embedded Transcoder," ACM Multi-Media 2001, September 30 – October 5, 2001, Ottawa, Canada, pp. 617-620.

[10] Javed I. Khan and Seung S. Yang, Resource Adaptive Nomadic Transcoding on Active Network, International Conference of Applied Informatics, AI 2001, Feburary 19 – 22, 2001, Insbruck, Austria.

[11] Javed I. Khan, Seung S. Yang, Made-to-order Custom Channel for Netcentric Applications over Active Network, Proc. Of International Conference on Internet and Multimedia Systems and Applications, IMSA 2000, November 20-23, 2000, Las Vegas, U.S.A., pp. 22-26.

[12] Bhattacharjee, S., Kenneth L. Calvert, and Ellen W. Zegura, "An Architecture for Active Networking," High Performance Networking' 97, White Plains, NY, April 1997 [also available at **http://www.cc.gatech.edu/projects/canes/papers/anarch.ps.gz**, October 98]

[13] Tennehouse, D. L., J. Smith, D. Sincoskie, D. Wetherall and G. Minden., "A Survey of Active Network Research," IEEE Communications Magazine, Vol. 35, No. 1, Jan. 97, pp 80-86

[14] Wetherall, Guttag, Tennehouse, "ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols," IEEE OPENARCH'98, San Francisco, April 1998. [also available at http://www.tns.lcs.mit.edu/publications/openarch98.html]

[15] Steve Berson, Bob Branden, Steve Dawson, "Evolution of an Active Networks Testbed," Proceedings of the DARPA ActiveNetworks Conference and Exposition 2002, pp. 446-465, San Francisco, CA, 29-30 May 2002

[16] M. Sanders, M. Keaton, S. Bhattacharjee, K. Calvert, S. Zabele and E. Zegura, "Active Reliable Multicast on CANEs: A Case Study," In Proceedings of IEEE OPENARCH 2001.

[17] Alex Galis, Bernhard Plattner, Eckhard Moeller, Jan Laarhuis, Spyros Denazis, HuiGuo, Cornel Klein, Joan Serrat, George T Karetsos, and Chris Todd, "A Flexible IP Active Networks Architecture," The Second International Conference on Active Networks (IWAN), Hiroshi Yasuda, Ed., Tokyo Japan, October 2000.

[18] C. Cook, K. Pawlikowski, and H. Sirisena, "Ants: A toolkit for building and dynamically deploying network protocols," IEEE OPENARCH 02, New York, NY, June 2002.

[19] M. Hicks, P. Kakkar, J. Moore, C. Gunter and S. Nettles, "PLAN: A Packet Language for Active Networks," Proceedings of the International Conference on Functional Programming (ICFP'98), September, 1998.

[20] J. T. Moore and S. M. Nettles, "Towards Practical Programmable Packets," Proceedings of the 20th Conference on Computer Communications (INFOCOM), IEEE , Anchorage, Alaska, April, 2001.

[21] S. da Silva, D. Florrissi, and Y. Yemini, "Composing Active Services in NetScript." DARPA Active Networks Workshop, Tucson, AZ,March 9-10, 1998.

[22] Daniel Reed, "Grids, the Teragrid, and Beyond," IEEE Computers, January 2003, pp. 62-68.

[23] I. Foster, C. Kesselman, J. Nick, S. Tuecke, "Grid Services for Distributed System Integraiton," IEEE Computer Magazine, 35(6), pp. 37-46, June 2002.

[24] Common Object Request Broker Architecture (CORBA) group, URL: **http://www.corba.org**

[25] Steve Berson, "A Gentle Introduction to the ABone," (ISI), OPENSIG 2000 Workshop, Napa, CA, 11-12 October 2000. http://www.isi.edu/abone/intro.html

[26] Andrew T.Campbell, etc., "A Survey of Programmable Networks," Computer Communication Review, vol.29, no.2, pp.7-23, Apr.1999.

[27] R. Boorstyn, A. Burchard, J. Liebeherr, and C. Oottamakorn, "Statistical service assurances for traffic scheduling algorithms," IEEE Journal on Selected Areas in Communications, Special Issue on Internet QoS, 2000.

[28] J. Redford, S. Sen, J. Dey, W. Feng, J. Kurose, J. Stankovic, and D. Towsley, "Online Smoothing of Live Variable-Bit-Rate Video," In 7th Workshop Network and Op. Systems Support for Digital Audio and Video, pp. 249-257, St. Louis, MO, May 1997.

[29] H. Zhang and D. Ferrari, "Rate-Controlled Static-Priority Queueing," In Proceedings of IEEE INFOCOM'93, pp. 227-236, San Francisco, CA, March 1993.

[30] N. Argiriou and L. Georgiadis, Channel Sharing by Rate-Adaptive Streaming Applications, IEEE INFOCOM'02, New York, June 2002.

[31] Jon C. R. Bennett, Kent Benson, Anna Charny, William F. Courtney, Jean-Yves LeBoudec, "Delay Jitter Bounds and Pcket Scale Rate Guarantee for Expedited Forwarding," IEEE INFOCOM'01, Anchorage, Alaska, April 2001.

[32] Donald L. Stone and Kevin Jeffay, "An Empirical Study of Delay Jitter Management Policies," Multimedia Systems Journal, volume 2, number 6, pp. 267-279, January 1995.

[33] Y. Mansour and B. Patt-Shamir, "Jitter Control in QoS Networks," 39th Annual IEEE Symposium on Foundations of Computer Science, pp. 50-59, October 1998.

[34] W. Ma, B. Shen, and J. Brassil, "Content Services Network: The Architecture and Protocols," Proceedings of the WCW'01, Boston, MA, June 2001.

[35] K. Anagnostakis, M. Greenwald, and R. Ryger, "cing: Measuring Network-Internal Delays using only Existing Infrastructure," Proceedings of the 22nd Annual Joint Conference of IEEE Computer and Communication Societies, INFOCOM 2003, San Francisco, CA, April 2003

[36] K. Anagnostakis, M. Greenwlad, S. Ioanndis, A. Keromytis, and D. Li, "A Cooperative Immunization System for an Untrusting Internet," Proceedings of the

11th IEEE International Conference on Networks, ICON 2003, Sydney, Australia, September 2003

[37] D. Wetherall, "Service Introduction in an Active Network," Ph. D. Thesis, MIT/LCS/TR-773, February 1999.

[38] Odyssey User's Guide, Version 1.3, URL http://www.cs.gatech.edu/project/canes

[39] A. Campbell, H. Meer, M. Kounavis, K. Miki, J. Vicente, and D. Villela, "A Survey of Programmable Networks," ACM Computer Communications Review, April 1999

[40] Katie Hafner and M. Iyon, Where Wizards Stay up Late: the Origins of the Internet, Simon and Schuster, New York, 1996.

[41] Peter Forman and Robert W. Saint, Creating Convergence, Scientific American, November 2000, pp.10-15, URL http://www.sciam.com/2001/0101issue/0101stix.html

[42] Javed I. Khan and Seung S. Yang, Systems overview of Active Subnet Diffusion Transcoding System for Rate Adaptive Video Streaming (NetAVT v2.0), Technical Report: 2001-11-04, Kent State University, [available at URL **http://medianet.kent.edu/technicalreports.html**, also mirrored at **http://bristi.facnet.mcs.kent.edu/medianet**]

[43] Javed I. Khan, Seung S. Yang, et al., Architecture Overview of Motion Vector Reuse Mechanism in MPEG-2 Transcoding, Technical Report: 2001-01-01, Kent State University, [available at URL **http://medianet.kent.edu/technicalreports.html**, also mirrored at **http://bristi.facnet.mcs.kent.edu/medianet**]

[44] Javed I. Khan and Seung S. Yang, Architecture Overview of Medianet Multiprocessor Transcoder, Technical Report: 2000-08-01, Kent State University, [available at URL http://medianet.kent.edu/technicalreports.html, also mirrored at http://bristi.facnet.mcs.kent.edu/medianet]

[45] Javed I. Khan and Seung S. Yang, Medianet Active Switch Architecture, Technical Report: 2000-01-02, Kent State University, [available at URL http://medianet.kent.edu/technicalreport.html, also mirrored at http://bristi.facnet.mcs.kent.edu/medianet]

Demonstration

- Resource Adaptive Netcentric System on Active Network: SONET, Active Networks Conference and Exposition, May 21-24, 2002, IEEE Computer Society Press, San Jose, CA.

- Self-Organizing Network Embedded Transcoder, ACM Multi-Media 2001, September 30 – October 5, 2001, Ottawa, Canada.

- Active Transcoder, DARPA Active Network Conference and Exposition, December 1-4, 2000, Atlanta, GA.