# Composition of Java-based Router Elements and its Application to Generalized Video Multicast

Yu Dong  David K. Y. Yau  John C. S. Lui

*Abstract*—We describe a software router capable of flexible service composition through plug and play of specialized Java software modules. These Java modules – previously developed for network simulation in the *J-Sim* project – are leveraged for actual deployment on our router through a *JSocket* class of objects. Our system provides significant software engineering benefits of simplified code development and safe composition/reuse of various router components. These benefits have proved highly useful in implementing new network services for emerging application needs. In particular, we present a paradigm of *generalized multicast* with application to large-scale video streaming. We detail the performance of our prototype implementation in terms of efficiency (when compared with a native C implementation) and its ability to satisfy the dynamic resource capabilities of a heterogeneous set of receiver end-systems including mobile handheld devices.

## I. Introduction

The Internet is expanding in scale and increasing in diversity. For example, mobile devices (laptops, handhelds, cell phones, etc.) are increasing in capability and are being connected to the network in larger numbers. New applications and services are being conceived and developed for the changing network environment.

To keep up with the evolving needs of the Internet, software-programmable routers are promising. They provide a highly flexible way to extend existing network protocols and configure protocol components according to the needs of emerging applications. With such a router, new services can be readily started by hot-swapping the configuration file of router modules, without shutting down the router. This feature is important for new service deployment and system resource management. Current composable routers predominantly support OS kernel modules implemented in C/C++. Click, for example, provides an element-based router architecture, in which elements (each being a Linux loadable module realizing a simple router function) can be configured for customized per-flow processing of packets. Deploying router functions as compiled C/C++ kernel modules has the advantage of being highly efficient.

Unfortunately, kernel modules – particularly modules that can be safely composed and reused – can be quite hard to develop. Weak data typing, lack of well-defined module interfaces, possibly intricate data dependencies between modules, platform dependence, absence of fault isolation, and an unfamiliar development and execution environment to most programmers, are some of the major stumbling blocks. The use of a modern object oriented language, like Java, will solve some of the problems. For example, Java is platform independent and has much stronger type checking than C/C++. By design, it explicitly disallows many of the "dangerous" operations, such as reckless memory management and pointer manipulations, that one can perform with C/C++. However, the unconstrained use of Java objects will *not* guarantee safe composition of these objects without unintended side effects. In particular, Java is susceptible to the *hyperspaghetti object and subsystem* phenomenon [13], in which objects can be so tightly and invisibly coupled together that it is difficult to extract classes in one system for reuse in another.

The *J-Sim* project [12] has implemented a software infrastructure for flexible large-scale network simulations. In *J-Sim*, protocol components are written in Java, but must conform to the requirements of the *autonomous component architecture* (ACA). In ACA, components communicate with one another by sending/receiving data at their entry points, called ports. The semantics of data exchanges are defined through certain *contracts* that are agreed upon at design time. A contract specifies the casuality of information exchange between components but not the types of the components participating in the exchange. A component is bound to contracts, rather than other components that interact with it. Binding contracts at design time and components at system configuration time eliminates the hyper-spaghetti phenomenon. Our work enables the use of *J-Sim* ACA components for actual deployment on operational routers, rather than for simulations.

### A. Our contributions

Our contributions in this paper are three fold. *First*, we realize a software router that can support the use of ACA components as building blocks of router services. This is done by integrating ACA into the CROSS/Linux software router [3] we have previously developed. Compared with existing modular routers, the target router will provide software engineering benefits of easier code development and safe composition and reuse of router components. *Second*, we leverage the significant code base of *J-Sim* protocol simulation modules for deployment on operational routers. This will greatly facilitate the development and experimentation of new network services while exploiting the rich body of complementary protocol functions developed in the Internet,

IntServ, and DiffServ contexts. In particular, we have applied our platform to generalized multicast of scalable video to a set of heterogeneous video receivers (e.g., desktops, laptops, and handhelds). IP multicast components, including IGMP and DVMRP, and other *J-Sim* components, are immediately leveraged on the router control plane. In addition, we have implemented an adaptation mechanism for video clients to *negotiate* the desired quality of video with appropriate routers. (The functions of duplicating and scaling received video packets are implemented in a separate module of the router forwarding plane.) *Third*, we have evaluated the performance of our prototype implementation in terms of efficiency (when compared with a native C implementation) and its ability to satisfy the dynamic resource capabilities of a heterogeneous set of video receivers (particularly the energy capacity of mobile handheld devices).

### B. Related work

Component-based synthesis of routing protocols has been advanced in x-kernel [8], and adopted to different extents in recent extensible routers [4], [1], [10]. The existing platforms, including x-kernel and Click, predominantly support compiled software modules implemented in C/C++, which can be quite hard to safely compose and reuse. Java-based software routers have been described in [11]. Our work in supporting *J-Sim* and ACA components goes further than simply using Java. Java by itself is susceptible to the hyper-spaghetti object and subsystem phenomenon, which can make it exceedingly difficult to extract objects from one system for reuse in another system.

Our generalized multicast service realizes adaptive streaming of scalable video to a set of heterogeneous receivers. While this is conceptually similar to previous work in scalable video streaming [5], our work is distinguished in two respects. First, we have carefully detailed the impact of our service in the increasingly important area of energy adaptation for mobile clients. We have also performed careful analysis of the energy impact on different system components, including display, network interface, and local processing, on a pocket PC. Second, our implementation demonstrates directly the feasibility of leveraging and reusing software components previously developed for a different goal (namely, network simulation) in actual router deployment. Not only does it validate the software engineering benefits and accompanying performance impact of our work, but it also opens up interesting possibilities of mixing matching software components for hybrid network simulation, emulation, and real deployment between machines. There has been similar work to extend the NS network simulator to support network emulation [2].

Luby et al. [7] have studied negotiation protocols that adapt application performance to available network bandwidth. Our work similarly targets resource-aware applications, but generalizes such adaptation to include receiver system resources such as CPU and energy capacities.

## II. ASSIMILATING *J-Sim* ROUTER COMPONENTS

*J-Sim* [12] is a compositional network simulation environment. It is built upon a component-based software architecture, called the *autonomous component architecture (ACA)*, and a packet-based network modeling framework, called the *extensible internetworking framework (EIF)*. In ACA, a software system is composed of a collection of components. ACA organizes the components with the interface of *Ports* to communicate with the outside. The ability to handle data in independent execution contexts, along with the fact that components are bound to one another only at system configuration time, enables a component to be autonomous (hence the name of the architecture), and be reused in other software systems.

The EIF model in *J-Sim* is built upon two layers: the *core service layer* (CSL) and the *upper protocol layer* (UPL). CSL includes the common internetworking services to the clients such as routing table services, interface services, etc. The component modules in UPL, such as application, transport, and routing protocols use the CSL services. Many of the *J-Sim* service components (e.g., routing protocols components in UPL, common services components in CSL) are in principle detailed enough to fully function in a real system as discussed in [12]. One of our tasks is to provide a *generic* mechanism for the simulation modules to run seamlessly on a router platform in practice.

An initial design issue for deploying *J-Sim* components is to enable the components to communicate over an actual network. We have defined a physical network access socket component, called *JSocket*, that interfaces the CSL of *J-Sim* with a physical network. JSocket is implemented in Java, and is designed to conform to the specification of an *active* ACA component. An ACA component is considered to be active if it is a *data source* that generates data inside the component and sends the data to other components through its connecting ports. JSocket may be considered a data source when receiving data from the actual network. JSocket can interface with other ACA components through its input and output ports. These ports can be connected to the *down port* of a Packet Dispatcher component of CSL to send/receive data. Hence, a pair of CSL components running on different machines are able to communicate with each other through their corresponding JSockets.

A second issue we faced is that in *J-Sim*, simulation messages are frequently sent between nodes in a simplified format (e.g., without full protocol headers, or by passing buffer pointers instead of the data themselves) for efficiency reasons. Such optimizations will not work for real communication between machines, since the message packets sent by the *J-Sim* service and routing protocol components will be known to *J-Sim* only, and are different from the standard routing protocol message formats. Thus, the messages cannot be recognized and used by a peer router talking the same protocol. To handle the problem, we incorporate a packet format conversion function into JSocket, which converts packets between the *J-Sim* formats and the standard network packet formats, including full header encapsulation. The packet conversion function, `Packet_convert`, is shown in Figure 1 which illustrates the internal structure of a JSocket.
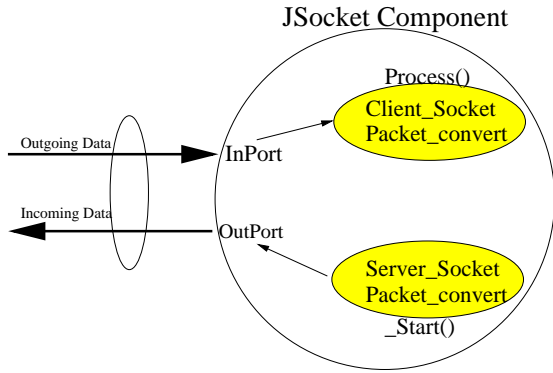
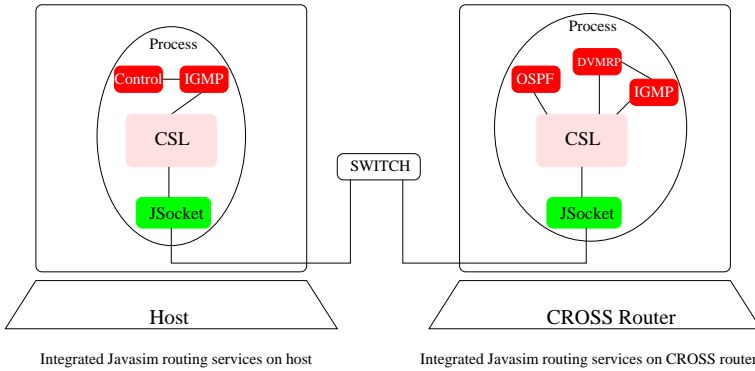Fig. 1. Internal structure of the JSocket component.



Fig. 2. *CROSS/Linux* routers configured with JSockets.

Also shown in Figure 1 are two special methods that make JSocket conform to the ACA specification for active components: (1) `Process()` that processes incoming data, and (2) `_start()` that generates outgoing data. In `Process()`, we use a client socket to send *J-Sim* messages generated by other *J-Sim* components to the actual network, after packet conversion. In `_start()`, a server socket listens for and receives data from the actual network. Received data are converted into *J-Sim* message objects. and then sent to other *J-Sim* components through JSocket's output port.

Apart from message conversion, a runtime environment in *J-Sim* automatically establishes connections between components according to certain simulation scripts. All the components of UPL and CSL are automatically instantiated and connected by the simulator runtime according to the description of a router node in a simulation script. In actual deployment, we do not have the simulation runtime. Therefore, we need to explicitly configure each component with other components, through the correct ports. Figure 2 illustrates an example router configuration that runs IP multicast as adapted *J-Sim* components between two actual machines.

## III. GENERALIZED MULTICAST SERVICE

Our router implementation is based on *CROSS/Linux* [3]. *CROSS/Linux* uses Click elements for service composition. Beyond Click, *CROSS/Linux* has significantly enhanced resource management capabilities, such as flow-based scheduling (Click scheduling is element based), fine grained preemption of packet processing, and QoS-aware provisioning

of various router resources (e.g., CPU cycles and network bandwidth). Whereas the forwarding plane of *CROSS/Linux* processes packet flows, its control plane runs supporting services such as routing (e.g., OSPF, RIP and DVMRP) and signaling (e.g., SIP and RSVP) daemons. We now explain the leveraging of certain *J-Sim* components (e.g., DVMRP and IGMP) on *CROSS/Linux* to support heterogeneous video multicast.

### A. Paradigm of generalized multicast service

As motivated in Section I, heterogeneous end systems have access to different system resources, including CPU capacity, network bandwidth, and energy capacity in the case of mobile devices. Large-scale data distribution (e.g., video streaming) over such heterogeneous networks is especially challenging because it must be scalable to a large number of receivers to be economically viable and, at the same time, be amenable to customization to cater to different customer needs. Scalability suggests that resources should be *shared* by receivers whenever feasible to amortize distribution costs. On the other hand, the need for customization suggests that, when receivers are heterogeneous, their payloads should be properly *differentiated* along the distribution pathways. To satisfy the dual goals of sharing and differentiation, we define a paradigm of *generalized multicast service* (GMS).

In GMS, a single transmission along a shared link can be used to satisfy a group of receivers, similar to traditional IP multicast. In addition, it allows *flow group specific operations* to be defined at branch points in a distribution tree. These features together enable receiver differentiation to be applied not only at the sender, but also at routers corresponding to strategic places inside the network infrastructure. In-network deployment gives two benefits. First, prior to differentiation, transmissions can be shared, claiming multicast economy of scale. Second, differentiation may be unforeseen at the sender, such as response to congestion on selected network paths, and so must be performed at network routing points.

### B. Implementation on CROSS/Linux by leveraging J-Sim *components*

We have implemented GMS for wavelet video streaming [5]. Each GMS client can request to receive a certain quality level of video consistent with its capability and current operating conditions (e.g., CPU load, remaining battery, and available network bandwidth). On the *CROSS/Linux* router control plane, we reuse the *J-Sim* IP multicast components, including DVMRP and IGMPv2, to maintain the routing and video quality information of members in a wavelet video multicast group. The IGMP component maintains the local multicast group information (e.g., member join/leave). DVMRP maintains the multicast routing information among routers. Both components rely on services from the CSL, like routing table management, packet dispatching, etc. Each protocol message augmented with video quality level information is sent to the PacketDispatch component of CSL, from which the packets are passed to JSocket. JSocket converts the messages

to actual protocol packets and sends them on to the destination machines.

In addition, we build a separate component for clients to negotiate the desired quality of video with appropriate routers. As part of IP multicast routing, each router maintains a routing table of the current set of downstream recipients (which is either a next-hop router or a client) for the multicast group. We augment the multicast routing table to include the video quality requested by each recipient. For example, our system supports Discrete Wavelet Transform (DWT) encoded video, which consists of a base layer of low frequency video information, and progressive enhancement layers of higher frequency information. The video is encoded in 33 progressive layers and therefore supports 33 video quality levels. The best quality level is level one consisting all the 33 layers. The lowest quality level of 33 has the base layer only. For a router, say $R$, assume that there are currently $n$ downstream recipients. We denote by $q_i$ the quality level requested by recipient $i, 1 \leq i \leq n$. If $R$ has an upstream router $S$ for the multicast group, it will then request a quality level of $\min\{q_i, 1 \leq i \leq n\}$ from $S$.

Using the routing and video quality information maintained by the control plane, the *CROSS/Linux* forwarding plane is responsible for forwarding received video data to each downstream recipient, after appropriate scaling of the video quality to that requested by the downstream. The function of duplicating and scaling received packets for the downstream is implemented as a standard *CROSS/Linux* element, called gmc, on the multicast data path of the forwarding plane.

The gmc element manages a local forwarding table that records the downstream recipients of a video being streamed, together with the video quality levels currently requested by each of the recipients. Whenever there is any change in the GMS routing table managed by the control plane (e.g., when clients join/leave or change their desired video quality level), gmc updates its local forwarding table accordingly through a kernel write file handler associated with GMS routing table. Also, a 7-bit tag in each video packet identifies the layer of the carried video information. By comparing the 7-bit tag of a video packet with the quality level requested by a downstream recipient, gmc can efficiently decide whether the packet should be forwarded to that recipient or not.

### C. Adaptation mechanism of GMS

In practice, the video quality level requested by a client should change according to the current network conditions, local system resource availability, etc. For example, network congestion or signal interference may cause excessive drop of video packets for a wireless client, and may lead to severe degradation of the video quality. Also, the battery of a PDA device may be running low and cannot support the highest quality for the whole duration of the video. It is important that GMS provides feedback control to monitor and adapt to these variations over time.

Our adaptation mechanism runs on the control plane of *CROSS/Linux*. At each end system, a negotiate_client process monitors in real time the available system resources –

currently, CPU utilization, battery power, and network bandwidth. When the estimated availability of a resource changes by more than a certain threshold, an end system may request a new video quality level from the routers. On receiving the new requested quality level, a negotiate_server daemon process running on a *CROSS/Linux* router will update the GMS routing table, which will in turn change the gmc forwarding table as discussed above. The new video quality information will also eventually be propagated to relevant upstream routers through DVMRP augmented to handle the additional quality information.

Consider, for example, adapting to available network bandwidth. For our wavelet video experiments, we have a per-video profile of the expected data rate for each quality level. The profile is contained in a table called rate_table accessible to the video clients. When an end system measures an achieved packet arrival rate (averaged over a time window currently set to be one second and denoted as sample_rate) that is lower than expected by the current quality level, say $p$, the end system will use rate_table to look up the highest quality level, say $q$, that can be supported by the measured actual packet arrival rate. Through negotiate_client, it will then request a quality level of $q$ ($q < p$) from the routing infrastructure. The algorithm for *reducing* the video quality if necessary is specified as algorithm_degrade:

```
1 algorithm_degrade(sample_rate){
2   for (level = 1; level <= MAX_LEVEL; level++){
3     if(rate_table[level]  < sample_rate){
4       return level;
5     }
6   }
7   return MAX_LEVEL;
8 }
```

More generally, we also need to adapt to *increased* available network bandwidth (e.g., when congestion eases). This will require us to discover the extra available bandwidth, and to improve the requested video quality accordingly. Algorithm algorithm_adapt is designed to perform this more general adaptation. It is called when a new sample_rate value is obtained. The algorithm calculates the difference, diff, of sample_rate minus the expected data arrival rate. If the difference is negative and exceeds a threshold value, we conclude that there is not sufficient bandwidth to sustain the current video quality, and use algorithm_degrade to reduce the quality of the requested video. Otherwise, if we record sufficient bandwidth for more than MAX_DELAY sample periods, we optimistically assume that there is extra bandwidth available, and attempt to achieve the highest "ideal" video quality, ideal_level, desired by this client. If this ideal quality cannot be sustained, then algorithm_degrade will soon be triggered again to reduce the video quality. Otherwise, the client enjoys its desired video quality, and no further control actions are needed.

```
1 algorithm_adapt(sample_rate){
2   diff = sample_rate - rate_table[current_level];
3   if (diff < 0) {
4     if (|diff| < threshold *
           rate_table[current_level]) {
5       if (recover_delay++ < MAX_DELAY) {
6         new_level = current_level;
7       } else {
8         new_level = ideal_level;
```

```
9            recover_delay = 0;
10       }
11    } else {
12       new_level = algorithm_degrade(sample_rate);
13    }
14 } else {
15    if (recover_delay++ < MAX_DELAY) {
16       new_level = current_level;
17    } else {
18       new_level = ideal_level;
19       recover_delay = 0;
20    }
21 }
22 return new_level;
23 }
```

## IV. EXPERIMENTAL RESULTS

We report experimental results using our system prototype to quantify the performance of our router and the GMS service. Our experimental router is a Pentium III/866 MHz machine with 128 MBytes of RAM and running Linux RedHat 2.2.4. All the *J-Sim* components run with IBM JDK 1.3 for Linux. The version of *J-Sim* used in the experiments is v1.0. The *CROSS/Linux* router version is v1.0. End systems include Sun Ultra-10s running SunOS 5.6, Pentium III PCs running Linux 2.2.4, iPAQ H3600s running Familiar Linux 2.4.18. Because of limited space, we will only present selected performance results; further results can be found in [6].

### A. Microbenchmark performance of J-Sim components

The easiest way to implement JSocket is to use the default Java TCP socket API to encapsulate *J-Sim* routing messages. However, since most routing protocols run at the IP layer, TCP encapsulated routing messages will not work with standard peer routers. We have therefore implemented JSocket in Java native methods using raw sockets to convert *J-Sim* routing messages into their standard formats. We aim to quantify the processing overhead of such packet conversion.

Our experiments measure the round trip time (RTT) of IGMP messages sent between an IGMP client and an IGMP server. The time is measured as the delay between when the client sends a join message to the server and when the server sends back a reply. To factor out the effect of the network link, we run both client and server as separate processes on the same machine, and connect the two through a loop-back IP address. We compare the achieved RTT of a C implementation with two Java implementation versions: one using original Java TCP sockets and another using our implementation of native raw sockets. For Java TCP socket, the default configuration is with Nagle's algorithm enabled. We have also experimented with Nagle's algorithm disabled to greatly reduce the delay for small message transfers as in our experiments. Our delay measurements are obtained with a native Java method, called `gethrtime()`, that we implemented to return timestamps in $\mu$s resolution and with an overhead of about 60 ns.

Breakdown of the RTT time into different components is shown in Table I. Reported numbers are averages over 1000 request-reply messages sent back to back between client and server.

From Table I, the Java implementation with raw sockets is significantly more efficient than the Java implementation with TCP sockets. It achieves an RTT slightly more than twice that of the C implementation. There are two reasons why the Java implementation is not quite as efficient as C: (1) the overhead induced by JNI (the package to create Java native methods) that copies the IGMP packet data between the Java runtime space and the native method runtime space, and (2) the "server processing delay" for the Java IGMP server implementation is higher than its C counterpart (see Table II). However, we believe that the achieved efficiency is adequate for a control service like IGMP, and is a reasonable price to pay for the software engineering advantages of ACA components.

For the IGMP server processing overhead, the breakdown is shown in Table II. The time of "receive IGMP packet" is only measured for the default Java implementation. It is the time to retrieve an IGMP packet from the `DataInputStream` of the Java socket after the TCP connection is accepted. "Process IGMP data" gives the time for the IGMP server to process the IGMP data and extract group membership and video quality level information after a raw IP packet has been received. "Update routing table" is the time to update the kernel multicast routing table interfacing with the `gmc` element on the forwarding plane. The time used to send back a reply to the IGMP client is given in "Send reply".

### B. Impact of GMS on video playback

A wavelet video decoder runs at each end system to play back scaled video forwarded by a CROSS/Linux router. We demonstrate how GMS may impact video playback at a receiver, especially the energy capacity for the mobile handheld devices.

We expect some GMS clients to be mobile devices, for which energy is an important system resource. We measure the energy consumption of wavelet video playback at different quality levels, when running on an iPAQ H3600 pocket PC fitted with an IEEE 802.11b wireless interface card. For the experiments, the video playback rate is fixed at 3 frames/second, to match the relatively limited processing capacity of the iPAQ, which does not have a hardware floating point unit to perform calculation-intensive image processing.

The energy used, $E$, in playing back a video is then calculated as:

$$E = I_{avg} \times V_{power} \times Time_{play} \tag{1}$$

where $I_{avg}$ denotes the average current drawn, $V_{power}$ denotes the voltage of the power supply, and $Time_{play}$ denotes the playback duration that is always 100 seconds in this experiment.

The total energy consumption in Table III shows the reduction in energy consumption as we go from high to low video quality. We notice that at level one, the iPAQ is actively processing data for a larger fraction of the playback time, whereas for level 33, the iPAQ is idle longer between playback of consecutive frames. This explains the energy reduction between different quality levels.

There are three major consumers of energy during video playback: the video display, the network interface, and iPAQ local processing (CPU, memory, etc), denoted as $E_D$, $E_N$,

and $E_C$, respectively. We have further estimated the relative energy use of these components during video playback. These estimates are given in Table III. For a baseline comparison, the row labeled "idle" corresponds to the estimates when the iPAQ is simply turned on but not doing video playback. From the table, notice that much of the energy saving as we go from high to low video quality is a result of reduced processing by the network interface.

To get an idea of the maximum energy saving that can be *potentially* achieved, we may deduct the idle energy use for the three components from the corresponding energy use during playback at the different quality levels. By doing so, we compute the percentage energy saving achieved over level one. The results in Table III show that much of the energy saving would still come from the network, but significant energy saving would then be possible for the other two components also.

## C. End-to-end GMS adaptation performance

We measure the *end-to-end* performance of GMS adaptation. We use the *traffic controller* (TC) as a tool to limit the bandwidth of end systems to emulate network congestion. The experimental setup is shown in Figure 3. In the setup, cordoba is the wavelet video server, sevilla is a *CROSS/Linux* router supporting GMS, cadiz is a machine running TC to limit its egress bandwidth when forwarding the scaled wavelet video destined for madrigal, the video client.
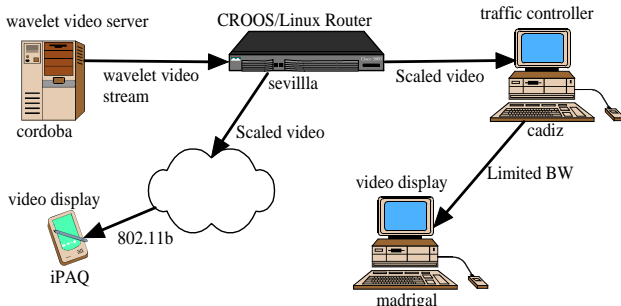
Fig. 3. Experimental setup for adaptation mechanism.

Our adaptation strategy allows clients to decide the video quality level desired locally, without relying on the routers (i.e., the approach is client-driven). The advantages are: (1) reduced control messages – end systems only send requests when necessary, without the need for periodic *heartbeat* messages that inform the routers of the available bandwidth, and (2) reduced load on routers, each of which may be shared by a large number of clients.

In the experiment, the client initially requests a video quality level of one. The playback duration is 20 seconds for total 300 frames. (The video frame rate is 15 frames/s.) The quality can be sustained when the network bandwidth is not restricted (100 Mbit/s). Then, we use TC to limit the available bandwidth to 100 Kbit/s. Fig. 4 profiles the achieved PSNR (Peak Signal Noise Ratio, a higher value of which means less distortion of image) with and without the adaptation mechanism, immediately after the reduction of
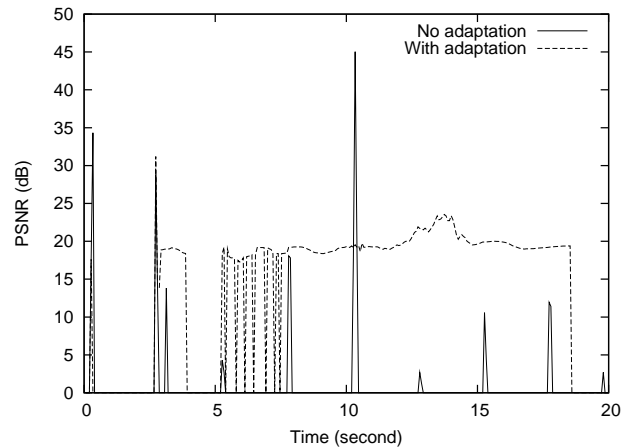
Fig. 4. PSNR profile of video playback with available bandwidth of 100 Kbit/s, with and without adaptation.

network bandwidth to 100 Kbit/s. Notice that in the case of no adaptation, the video playback essentially fails (only 19 frames are successfully displayed with average PSNR of 0.93 dB). With adaptation, however, the client finds out about the bandwidth reduction after a delay of about 1.5 seconds and changes the quality level to 26. The new video quality is still too high to be sustained. About one second later, the client further adjusts the video quality level to 33, which then allows the video to be played back at an appropriately degraded quality. As a result, 213 frames are successfully displayed, with an average PSNR of 13.78 dB. The sudden drop of PSNR after 18 seconds happens because the bandwidth requirement of the video increases due to a scene change around 18 seconds. The required bandwidth is significantly larger than the current available network bandwidth, and causes the video quality degradation. Such a problem could be mitigated by certain image processing techniques of error concealment, but is beyond the scope of the present work.

## V. CONCLUSION

We have presented the design and implementation of a software router supporting Java-based ACA components as building blocks of router services. The use of ACA components simplifies code development, achieves platform independence, and facilitates module composition and reuse. In particular, through our JSocket design, we are able to provide a generic means of adapting and leveraging a significant base of Internet, IntServ, and DiffServ protocol simulation modules for actual deployment on our router platform. As a case study, we have reused the *J-Sim* DVMRP and IGMP modules in realizing a generalized multicast service (GMS) for adaptive video streaming to heterogeneous users. We show that the use of *J-Sim* components, while not quite as efficient as native C implementations, has acceptable performance for control plane services. We also detailed how GMS may impact video clients in terms of CPU processing, network bandwidth, and energy requirements. Our results show that wavelet video scaling can be highly effective in gracefully matching the video playback quality to varying availabilities of these resources. Finally,

GMS allows the video quality to be automatically negotiated in response to changing system conditions. In future work, we aim to further detail our router's performance (transfer throughput, detailed processing overheads for each component, etc) and to realize a component-based software environment for hybrid simulation, emulation, and synthesis of network protocols.

## VI. Acknowledgements

## References

[1] D. Descaper, Z. Dittia, G. Parulkar, and B. Plattner. "Router Plugins: A Software Architecture for Next Generation Routers," In *Proc. ACM SIGCOMM*, Vancouver, Canada, Sept. 1998.

[2] K. Fall. "Network Emulation in the Vint/NS Simulator," In *Proc. Fouth IEEE Symposium on Computers and Communications (ISCC'99)*, Red Sea, Egypt, June 1999.

[3] P. Gopalan, S. C. Han, D. K. Y. Yau, X.Jiang, P. Zaroo, and J. C. S. Lui, "Application Performance on the CROSS/Linux Software-Programmable Router," CS TR-01-019, Purdue University, West Lafayette, IN, November 2001.

[4] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The Click Modular Router," *ACM Transactions on Computer Systems*, 18(3):263-297, August 2000.

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

[6] Y. Dong, D. K. Y. Yau, and J. C. S. Lui, "Composition of Java-based Router Elements and Its Application to Generalized Video Multicast", Technical Report, Dept of Computer Science, Purdue Univeristy, West Lafayette, IN, January 2004.

[7] M. Luby, V. K Goyal, S. Skaria, and G. B. Horn, "Wave and Equation Based Rate Control Using Multicast Round Trip Time," In *Proc. ACM SIGCOMM 2002*, Pittsburgh, PA, August 2002.

[8] S. W. O'Malley and L. L. Peterson. "A Dynamic Network Architecture," *ACM Transaction on Computer Systems*, 10(2):110-143, May 1992.

[9] C. Pfister and C. Szyperski, "Why Objects Are Not Enough," In *Proc. of the First International Component Users Conference (CUC'96)*, 1996.

[10] T. Spalink, S. Karlin, L. Peterson, and Y. Gottlieb. "Building a Robust Software-based Router Using Network Processors," In *Proc. of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2001. to appear.

[11] P. Tullmann, M. Hibler, and J. Lepreau. "Janos: A Java-oriented OS for Active Networks," *IEEE JSAC*, March 2001.

[12] H. Tyan and C. Hou, "Design, Realization, and Evaluation of A Component-based, Compositional Network Simulation Environment," *2002 SCS Western Multiconference on Computer Simulation – communication networks and distributed systems modeling and simulation conference*, January 2002.

[13] B. F. Webster. "Pitfalls of Object-Oriented Development," M&T Books, New York, 1995. ISBN 1-55851-397-3.

## VII. Authors' Biographies

**Yu Dong** received the B.E. degree from the School of Automation and Information Engineering, University of Science and Technology Beijing (USTB), China in 1995, the M.E. degree in Information System Engineering from Osaka University, Japan in 2001, and the M.S. degree in Computer Science from Purdue University, U.S.A. in 2003. He is currently working towards his Ph.D degree at Purdue. His research interests are in networking, including wireless ad-hoc and overlay networks, multimedia systems, and VLSI design. Since 1999, he has been a student member of the IEEE.

**David K. Y. Yau** received the B.Sc. (first class honors) degree from the Chinese University of Hong Kong, and the M.S. and Ph.D. degrees from the University of Texas at Austin, all in computer sciences. From 1989 to 1990, he was with the Systems and Technology group of Citibank, NA. He was the recipient of an IBM graduate fellowship, and is currently an Associate Professor of Computer Sciences at Purdue University, West Lafayette, IN. In 2004, he is on the faculty of the Department of Computer Science and Engineering, The Chinese University of Hong Kong. He received an NSF CAREER award in 1999, for research on network and operating system architectures and algorithms for quality of service provisioning. His other research interests are in network security, value-added services routers, and mobile wireless networking. David is a member of the ACM and IEEE. He serves on the editorial board of the *IEEE/ACM Transactions on Networking*.

**John C. S. Lui** received his Ph.D. in Computer Science from UCLA. He worked in the IBM T. J. Watson Research Laboratory and in the IBM Almaden Research Laboratory/San Jose Laboratory before taking up an academic position at the Chinese University of Hong Kong. Currently, he is leading a group of research students in the Advanced Networking and System Research Group. His research encompasses both systems and theory. His current research interests are in theoretical/applied topics in data networks, distributed multimedia systems, network security, OS design, mathematical optimization, and performance evaluation. John received the Vice-Chancellor's Exemplary Teaching Award in 2001. He is an associate editor of the Performance Evaluation Journal, a member of the ACM, a senior member of IEEE, and an elected member of the IFIP WG 7.3. John serves as the TPC co-chair of ACM Sigmetrics 2005. His personal interests include films and general reading.

TABLE I

BREAKDOWN OF IGMP REQUEST RTT IN $\mu$S.

|  | Java | Java w/o Nagle's | Java raw socket | C |
|---|---|---|---|---|
| Send IGMP packet | 31 | 509 | 117 | 61 |
| IGMP server processing | 9995 | 131 | 79 | 23 |
| Average RTT | 10026 | 640 | 186 | 84 |
| Slowdown over C | 119.4 | 7.6 | 2.2 | 1 |

TABLE II

BREAKDOWN OF IGMP SERVER PROCESSING OVERHEAD IN $\mu$S.

|  | Java | Java w/o Nagle's | Java raw socket | C |
|---|---|---|---|---|
| Receive IGMP packet | 9881 | 45 | - | - |
| Process IGMP data | 22 | 21 | 19 | 11 |
| Update routing table | 53 | 42 | 44 | 4 |
| Send reply | 22 | 16 | 12 | 6 |

TABLE III

ENERGY CONSUMPTION AND RELATIVE SAVING FOR EACH VIDEO QUALITY LEVEL IN J.

|  | $E_N$ (% saving) | $E_D$ (% saving) | $E_C$ (% saving) | $E_{Total}$ |
|---|---|---|---|---|
| Level 1 | 471.7 (0) | 55.5 (0) | 273.9 (0) | 801.1 |
| Level 5 | 437.5 (21.5) | 50.9 (30.8) | 272.8 (9.2) | 761.2 |
| Level 15 | 366.7 (65.9) | 49.8 (38.3) | 271.8 (17.5) | 688.3 |
| Level 25 | 337.8 (84.1) | 51.1 (29.5) | 271.3 (21.7) | 660.2 |
| Level 33 | 326.7 (91.1) | 47.8 (51.7) | 270.9 (25) | 645.4 |
| idle | 312.5 (-) | 40.6 (-) | 261.9 (-) | 615 |