A Framework-Based Approach to the Development of Network-Aware Applications

J. Bolliger and T. Gross

Abstract— **Modern networks provide a QoS (quality of service) model to go beyond best-effort services, but current QoS models are oriented towards low-level network parameters (e.g., bandwidth, latency, jitter). Application developers, on the other hand, are interested in quality models that are meaningful to the end-user and therefore struggle to bridge the gap between network and application QoS models. Examples of application quality models are response time, predictability, or a budget (for transmission costs).**

Applications that can deal with changes in the network environment are called *network-aware*. A network-aware application attempts to adjust its resource demands in response to network performance variations. This paper presents a framework-based approach to the construction of network-aware programs. At the core of the framework is a feedback loop that controls the adjustment of the application to network properties. The framework provides the skeleton to address two fundamental challenges for the construction of network-aware applications: (i) how to find out about dynamic changes in network service quality and (ii) how to map application-centric quality measures (e.g., predictability) to network-centric quality measures (e.g., QoS models that focus on bandwidth or latency).

Our preliminary experience with a prototype network-aware image retrieval system demonstrates the feasibility of our approach. The prototype illustrates that there is more to network-awareness than just taking network resources and protocols into account and raises questions that need to be addressed (from a software engineering point of view) to make a general approach to network-aware applications useful.

Keywords— **Adaptive applications, network-aware computing, software construction, frameworks, QoS.**

I. Introduction

MANY applications use networks to provide access to remote services and resources. However, in today's networks, users experience large variations in performance; e.g., bandwidth or latency may change by several orders of magnitude during a session.

Such dramatic changes are observed in mobile environments (where a user moves from one location to another) as well as in stationary environments (where other network users cause congestion). Variations in network performance are a problem for applications since they result in unpredictable application behavior. Such unpredictability is annoying—e.g., if a user looks through an on-line catalogue, a certain bandwidth must be continuously available if the system wants to display images at the speed expected by the user, or when congestion frustrates the user to the point that the software becomes unusable.

To bridge the gap between network reality and application expectation, i.e. to cope with the performance variations and to provide for a certain predictability of the application behavior, a number of researchers have proposed the development of *network-aware* applications. The basic idea is to allow an application to adapt to its network environment, e.g., by trading off the volume (and with it the quality) of the data to be transferred and the time needed for the transfer. That is, the application responds to a drop in bandwidth by reducing its demands on the networks, and increases its demands when there are additional resources.

To develop a meaningful approach to such adaptation, we must understand the realities of today's network architectures and the dynamics of the services provided. There can be many reasons for the variation in network performance. Some of the reasons are inherent (e.g., for mobile wireless communication), others are caused by the tremendous demand that always seems to outgrow any capacity improvement. In response to this situation, modern networks are beginning to move away from the best-effort service model to QoS models that allow the definition of quality metrics based on a variety of parameters. Unfortunately, current QoS models are oriented towards low-level network parameters (e.g., bandwidth, latency, jitter). Application developers, on the other hand, are interested in quality models that are meaningful to the end-user, such as response time. Thus, network awareness includes mapping application-centric quality measures (e.g., predictability) to network-centric quality measures and vice versa.

Another motivation for network awareness is to avoid the distinction between different application *modes*. For example, some image retrieval systems distinguish between a preview (or browse) mode, where only thumbnails are provided, and a mode of higher quality image delivery. Avoiding the concept of a mode simplifies implementation of the application components and allows the system to dynamically take advantage of available resources. A user on a high-bandwidth local area network does not have to live with a thumbnail-sized view that is statically defined and optimized for users accessing the image server across a (slow) wide-area network. A time limit parameter that controls how long a client is willing to wait provides enough flexibility to toggle implicitly between the browsing and the high-quality mode. Applications may need to adapt either at start-up time or dynamically during the course of a session or both.

There exist a number of network-aware applications, in particular from the realm of multimedia [22], [2]. However the solutions to the problem of network variability adopted by this class of applications are often tailored to the specific needs of

J. Bolliger is with the Department of Computer Science, Swiss Federal Institute of Technology (ETH), Zürich, Switzerland. E-mail: bolliger@inf.ethz.ch.

Effort sponsored in part by ETH Polyprojekt 41-2641.5.

T. Gross is with the Department of Computer Science, ETH, Zürich, Switzerland, and with the School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213. E-mail: thomas.gross@cs.cmu.edu.

Effort sponsored in part by the Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0287. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

an individual application or a specific programming model [41], and there exists no general approach to develop network-aware applications for other application domains. As network awareness continues to be an important aspect of application development, the need arises to identify and provide a general approach to build network-aware systems.

We propose to use *frameworks* as an approach that encapsulates (and integrates solutions to) the problems of adapting an application's behavior to the availability of network resources. A framework provides a basic solution to a class of problems; clients of the framework employ the basic structure by extension, i.e. they provide concrete methods where the framework relies on abstract methods [39]. So to build network-aware applications by extending a framework, we must develop the overall structure, which is the foundation for a framework, as well as the specific extensions that result in a real system, as has been done in other application domains where frameworks have proven useful.

The paper is organized as follows: Section II discusses issues related to the problem of network-awareness. Section III introduces the basic structure of our framework and the service model supported; Sections IV and V provide a detailed description of the methods employed to obtained information about network resource availability and the strategies used to adapt to changes in service quality respectively. After presenting performance measurements in Section VI we summarize related work and present our conclusions.

II. Network- and system-awareness

Networks are just one of the many resources employed by an application. The model of a network-aware application emphasizes the crucial role of the network connection: in many cases, the network is on the critical path, and performance problems in the network are the cause of the degradation of application performance. However in other cases, a system is bottlenecked by other components, e.g., the transfers across a local bus or from the disks, or the amount of computation. (Some experimental systems support a QoS model for internal transfers [15], [10], [9].) If application performance is limited by parts other than the network, then such an application should not be network-aware but *system-aware*, i.e. it should be able to adjust its behavior in response to other aspects of the system (response time, disk I/O latency, bus bandwidth, etc.). In the context of this paper we focus on the concept of *network-awareness* implying that an application's behavior is primarily controlled by the availability of network resources, but we point out where an application must go beyond network issues. System-awareness is especially important if an application wants to trade off communication and computation, i.e. an application may adjust to network changes by computing, e.g., compression. In such cases it is important to make sure that the computation overhead is not worse than the network overload.

For our discussion of network service quality awareness we concentrate on unicast request-response type communication between clients and servers, where the traffic in at least one direction can be described as *bulk-transfer* type network traffic. This traffic pattern makes up a large fraction of application traffic patterns observable in today's networks [5], [29].

A. Reservation vs. adaptation

Another approach to couple a service quality-aware application to a network is to allow the application to reserve network services in advance [40]. We do not discuss the relative benefits of either approach since in practice, both of them coexist (and continue to do so for a long time). Some network architectures (or their implementations) may not support reservations at all or may support them only to a limited degree (either by choice or due to implementation faults), and although future versions of popular protocol suites may support reservations, not all sites will run the most recent software. Furthermore, as network providers attempt to develop usage-based charging schemes, there will be financial incentives to restrain applications from uncontrolled use of network resources. (Today's networks have really two aspects that make adaptivity unattractive: there is almost no usage-based charging, and what is worse, the most aggressive applications are often rewarded with the largest share of the bandwidth pie [12].)

In a reservation-based approach an application must address the two issues of (i) how to find out what and how much to reserve (e.g., given some limit on the costs) and (ii) how to adjust to meet the confirmed reservation, which may be less than the application has asked for.

From a software engineering point of view, however, both techniques require the same software technology: an application must be able to adjust its resource demands, either to meet a limit imposed by a reservation or to meet some constraints imposed by the network. In either case the application must be adaptive.

B. Quality

The objective of network-awareness is to allow an application to be sensitive to changes in the network environment with the goal of maximizing user-perceived quality. In our context, quality means "conformance to a standard or a specification". Our focus on system-awareness means that we are interested in "the totality of features and characteristics of a product or service that bears on its ability to satisfy given needs" [17].

Only the application (developer) knows what "quality" is. So we build an infrastructure for those applications that are interested in a quality-time tradeoff, i.e. applications that are willing to sacrifice some degree of quality in return for faster response time (or are willing to wait a little longer to get better results). So the central issue is that we must find a software structure that allows the application developer to specify what "quality" means in the context of a specific application.

III. Framework for network-aware applications

Before we can discuss a specific framework, we first want to lay out a roadmap for the kind of interaction that is possible or profitable between an application and the network. The framework then provides, for some class of applications, a way to structure their interaction with the network through extending the framework. We start with principles of application-network interaction; these principles stem from our experience with various application projects and reflect study and rework involved while factoring possible framework structures. We illustrate the general principles with examples from a specific project, the Chariot (Swiss **A**nalysis and **R**etrieval of **I**mage **O**bjec**T**s) project [1], which is described in more detail in Section V-A. The objective of the Chariot project is to allow networked clients to search a remote image database. The Chariot system contains an adaptive image server that serves as proof-of-concept for the general ideas presented in the remainder of this paper.

A. Service model

Many networked applications using request-response type communication include a user (client) that requests a set of objects (images, texts, videos, byte code, etc.) from a remote site (server), which is responsible for retrieving the requested objects (from secondary storage) and delivering them to the client. The response usually has a larger volume than the request and dominates the transmission costs. In the following, we sometimes refer to such servers and clients as *sender* and *receiver* of the bulk-transfer, respectively.

A server accepts and acts upon request messages containing a list of *objects* to be retrieved (or computed) and some QoS-restrictions, where QoS-restrictions characterize the minimum *quality* tolerable for the objects delivered, the maximum quality that is beneficial for the user, and a limit T on the time allowed for processing the request and transmitting the response. The bounds on the quality may be (implicitly) imposed by the requester's processing or display capabilities. The application decides what kind of objects can be requested; *quality* is a property of a requestable object and must also be defined by the application.

Example (Chariot): Requestable objects are images or image sequences. The quality of an image is defined by the resolution, color depth, the image format (e.g., JPEG, GIF), a format-specific parameter, such as JPEG's compression factor [18], and a user-defined weighting of these image characteristics.

The server's task is to deliver all the requested objects to the client within time T, attempting to maximize the overall quality of the objects transmitted while respecting the QoS-restrictions. The range for dynamic adaptation to bandwidth availability is bounded by the minimal and maximal quality specified by the client. To quantify the task of the server, a *quality metric* must be defined by the application, e.g., as a weighted sum of the individual object qualities to be delivered. Weights for the quality metric may include the relative importance of an object in comparison to the other objects in the request list.

Example (Chariot): The weight of an image in the image request list is determined by a value for the similarity of the image with respect to a query image.

Such a network-aware server need not only dynamically adapt due to network service degradation (e.g., a drop in bandwidth), but should also try to opportunistically exploit extra bandwidth to deliver as many high quality objects as possible within time T.

Network-aware applications adhering to the service model above must address the following questions: (i) how to find out about dynamic changes in network service quality on the path from the sender to the receiver, and (ii) how to adapt the delivery process to such dynamic changes such that the objectives of the service model are met. Before we turn to each question in detail in Sections IV and V, we present a general structure for the type of network-aware application under consideration.

B. Application structure: software feedback control loop

A useful structure for network-aware applications using request-response type communication is a software feedback control loop, where the time left for the response—initially set to T—constitutes the command variable of the closed-loop control. The feedback driving the sender adaptation comprises information about the currently available bandwidth as obtained by mechanisms described in Section IV.

We focus on closed-loop control systems because they are in a position to deal with bursty applications. Other applications, e.g., those that deal with continuous media streams, may use a different control structure [2], [22]. We model sender-initiated adaptation in a closed-loop control system with the three phases *monitor* and *react* (P_{mr}), *prepare* (P_{prep}), and *transmit* (P_{trans}), as depicted in Fig. 1. The three phases work independently and share the list L of requested but not yet transmitted objects. P_{mr} is responsible for obtaining information (or feedback) about the available bandwidth, determining whether the amount of data to transmit must be reduced or whether it may be increased. In case adaptation is needed, the P_{mr} phase must decide which objects to adapt, which transformations to apply, and must then set the quality state of the objects according to these decisions. The term "transformation" refers here to any activity, including transfers, conversions, or computation. Once a (final) decision on the quality of an object to be delivered has been made, P_{prep} must transform the object to the quality assigned by P_{mr}. P_{trans} delivers completely prepared objects to the client.

Note that P_{mr} does not invoke transformations directly, but defers their execution to forthcoming phases P_{prep} to allow for "last-minute" adaptation. Furthermore, note that while P_{mr} may need to change the quality state of several objects at the same time, P_{prep} makes only one object ready for transmission at a time (on a uniprocessor).

IV. Feedback from the network

A central issue that determines the effectiveness of the control loop (and the frameworks built on this loop) is how it obtains information about the state of the network.

A. What does a network-aware application want to know?

With the aim to provide predictable service, i.e. response delivery within a specified amount of time T, an application ideally wants to know the network service quality, and in particular the bandwidth available for the time T. With a best-effort network service model, such as IP's and ATM ABR's [7], there is no way of getting such information in advance. Thus, all we can do is gather as much QoS-information about the past behavior as possible (and useful) and extrapolate future network behavior from the observed QoS-values.

We can distinguish two different application-relevant characteristics as far as bandwidth feedback is concerned: *bottleneck* bandwidth and *available* bandwidth [30]. The former gives an upper bound on how fast and how much an application may possibly transmit, while the latter gives an estimate on how fast the connection should transmit to preserve network stability, which is an issue of primary concern to congestion control mechanisms.

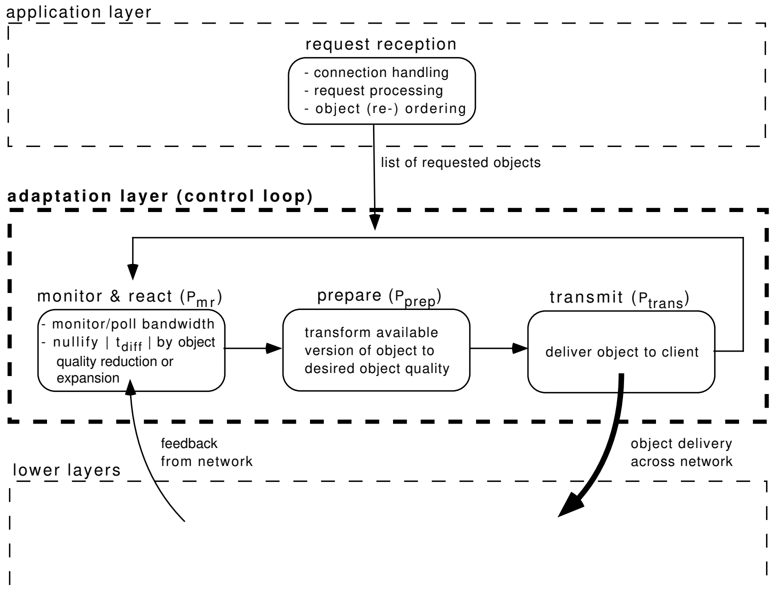


Fig. 1. Control-loop consisting of three phases *monitor* and *react* (P_{mr}), *prepare* (P_{prep}), and *transmit* (P_{trans})

While knowledge about the bottleneck bandwidth is useful in bounding the approximations for the bandwidth estimates used by a network-aware sender, information about the dynamics of the available bandwidth on the end-to-end network path is indispensable to enable timely adaptation of the volume of data to be transmitted.

B. Three approaches to obtaining feedback

This section discusses three approaches to obtain feedback about the characteristics and the dynamic behavior of an end-to-end network path. The distinction is based on the layering of the ISO/OSI-protocol stacks. The higher the layer providing the feedback, the less cooperation is required from network protocols on one side, but the less accurate and frequent will the feedback information be on the other side. Feedback about network service quality may be provided by:

Application-level QoS monitoring: A monitor assesses the dynamics of network service quality by measuring sender and receiver network quality parameters (e.g., packet interarrival times, bandwidth, etc.) and repeatedly exchanges the QoS-state between the peers, similar to the model proposed in RTP [35]. The timeliness and accuracy of the information depends on the averaging interval used for the computation of the QoS-values and the frequency of the QoS-state exchange. The monitoring approach provides only a *black box* view of the network and transport services. Therefore the sender has difficulties in distinguishing between service degradation caused by the network and degradation caused by the application or the end-system. E.g., a (temporarily) slow receiver may lead the sender to wrongly assume a network service degradation.

End-to-end transport-level congestion control: The goal of a congestion control algorithm is to operate at a connection's fair share of the bandwidth. To do so it must deploy mechanisms to find the bottleneck bandwidth and detect incipient congestion or network under-utilization. The *implicit* feedback that drives the adaptation of the sending rate may include the fraction of packets lost or measurements of delay variations, interarrival times of packet-pairs, etc.. Several benefits can be gained from making such transport-level feedback information transparent to a network-aware application: the feedback-loop is shortened and queuing unnecessary data for transmission can be avoided in times of congestion. Such information may help in bringing the application's behavior in line with the protocol's behavior, since the application has the same view of the network resources as the protocol. Furthermore, if the congestion control algorithm can make transparent its conclusions about the available bandwidth, an even tighter coupling between application and network can be achieved.

Network-level traffic management: Routers are most suited to fairly allocate resources among competing connections. Routers are the only authority capable of identifying and isolating misbehaving senders. Furthermore, routers are able to provide *explicit* feedback about their congestion state to the end-systems. Each router on an end-to-end path may generate feedback messages (either in binary form [31] or as an explicit rate information [7]). The feedback must be processed in the end-systems to find the available bandwidth used to control the sending rate.

It is important to note that the different layers may have different perceptions of the current network status since they employ different mechanisms to deal with exceptions such as loss events. However, as far as the estimation of available resources is concerned they all strive for a view as accurate as possible as it helps them avoid exception situations. Therefore, each layer may provide the information needed by a network-aware application, however, the lower the layer the more timely and accurate the information will be.

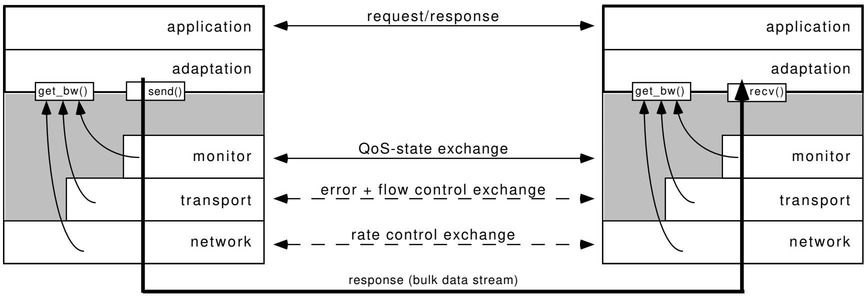


Fig. 2. Layered architecture of network-aware applications with the *adaptation* layer implementing the closed-loop control of Fig. 1.

C. Unified API

Although all three layers employ different feedback mechanisms, they all aim at finding the available network service quality to control an application's sending rate. Therefore, we devise a unified API for network service quality feedback in general and bandwidth feedback in particular that provides a network-aware application with the required information. As the application is interested in obtaining predictions about the bandwidth to be expected (or any other QoS-value) and an estimation for the reliability of the prediction, we extend a common transport protocol API, the socket API [37], by a function $get\_bw()$ that returns $bw(t)$ predicting the bandwidth for $t > now$, and $prob_{bw}(t)$, an estimate for the stability of the prediction[1]. Note that to ease framework development we provide the same QoS-interface at each layer in Fig. 2. Note also that both the *monitor*- and the *adaptation*-layer are logically part of the application.

As provision of dynamic network QoS-information is not the main topic of this paper we restrict our discussion to exemplifying how end-to-end congestion control information can be made transparent to an application through the API described. Our implementation of a (TCP-based) user-level transport protocol [4] distinguishes three high-level sender (congestion) states: start-up (slow start), congestion avoidance, and congestion recovery [19]. Each of the three state-classes (see State pattern in [14]) provides the function $get\_bw()$:

(i) The slow-start phase uses packet-pair probing to estimate the bottleneck bandwidth[2] bw_{max} and returns the function $bw(t) = min(bw_{max}, \frac{cwnd \cdot 2^{\lfloor t/RTT \rfloor}}{RTT})$, where $cwnd$ denotes the current congestion window and RTT stands for the (measured) round-trip time. In this phase, $bw(t)$ reflects slow-start's doubling of the bandwidth occupied every round-trip time, which is represented by the ratio of $cwnd$ and RTT. The exponential increase continues up to (at most) the network path's bottleneck capacity bw_{max}.

(ii) When the protocol is in the congestion avoidance state, i.e. when operating at the bandwidth effectively available, we deploy TCP Vegas-style network-path adaptation [3], and can therefore approximate $bw(t) \approx \frac{cwnd}{RTT}$, as changes to $cwnd$ are supposed to happen on a fairly large time-scale (multiples of the round-trip time).

(iii) In the congestion recovery state, which effects a rate halving, $bw(t)$ is modeled according to [21].

Congestion control's *use-it-or-lose-it* property [11] requires the sender to be almost constantly sending, otherwise the feedback may not be useful. Moreover, the issue of *dynamically* assessing the stability of end-to-end network-path characteristics is an open research question, which is why we refrain from discussing how to compute $prob_{bw}(t)$ here and refer to off-line studies on this topic [30].

V. Feedback loop and adaptation

As stated in the previous sections, the goal of a network-aware sender is to meet a user-specified bound on the delivery time by adapting the quality of the objects delivered to the measured/available network capacity. The adaptation process' objective must be to utilize the available resources as efficiently as possible and therefore to maximize the user-perceived quality within the bounds (time, bandwidth, and boundary conditions on quality) given. The following sections discuss in more detail the mechanisms deployed in our prototype network-aware system and elaborate on where and how application-specific information can/must be factored out of the software control system described to provide a reusable framework. However, before we turn to the framework structure and its interaction with an application, we briefly introduce the Chariot system as an example of the type of application that can be based upon this framework.

A. Chariot: sample framework instantiation

The objective of the Chariot project is to allow networked clients to search a remote image database. The Chariot system uses query-by-example to let a user formulate a query for similar images [1]. The low-level content (e.g., color and texture) of each image in the repository is extracted to define feature vectors, which are organized in a database index at the search engine. The core of the system (as depicted in Fig. 3) consists of a client (to handle user access to the image library), a search engine to identify matching images, and one or more *network-aware* servers, which deliver the images in the best possible quality, considering network performance, server load, and a client-specified delivery time. Physical separation of the image library index (in the search engine) from the image repository (in the server) facilitates distribution and mirroring of the library. The core components are connected by a coordination layer that isolates the details of network access and adaptation and gives each component a maximum of flexibility to take advantage of future developments.

[1] For the sake of brevity we only discuss bandwidth-related functions. Similar API-extensions exist for other QoS-parameters, such as delay or loss.

[2] This process is known as initial slow-start threshold (*ssthresh*) estimation [3], [16]. Note that standard TCP uses a statically defined *ssthresh* of 64 KBytes.

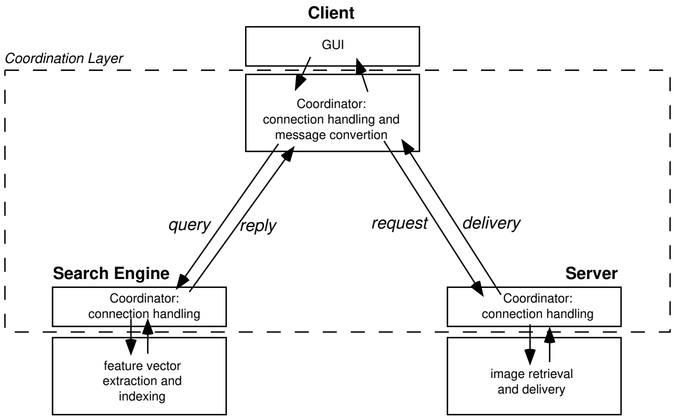


Fig. 3. Chariot architecture

It is the subsystem comprising client and the adaptive image repository that is relevant to our discussion of *network-awareness* and which serves as proof-of-concept for the ideas presented in this paper.

B. Monitor- and react-phase

Table I summarizes the terms and abbreviations introduced in the next sections. The discussion of application-specific information that is factored out of the control loop framework and which must be provided by the application (developer) always refers to the OMT-style [33] class hierarchy depicted in Fig. 4. The name of abstract classes, which are part of the framework, and abstract methods is shown in italics. Concrete classes provided by the application that instantiates the framework—Chariot in our example—are shaded. In the text we use "functional" notation. E.g., $foo(obj)$ or $bar(class)$ indicate that the method *foo* is invoked on object obj ($obj.foo()$ in our C++ implementation) or that the method *bar* is invoked from class *class* ($class::bar()$), respectively.

The *monitor*- and *react*-phase (P_{mr}) is the key phase in our framework. It is responsible for repeatedly obtaining feedback from lower protocol layers and deciding whether adaptation is required or not. The software control loop is part of the application and may be layered on top of a network monitor (see Fig. 2), from which it extracts feedback information about the available network service quality (e.g., bandwidth).

As P_{mr} is primarily interested in feedback about "relevant" changes in service quality, it either must deploy a polling policy to obtain feedback about the available bandwidth and assess the significance of a QoS change on its own, or it has to register with the monitor layer for asynchronous notification of QoS change events. Whether a change in network service quality is relevant is application-specific and depends on the granularity of the adaptation possible, the cost incurred by the adaptation mechanisms as well as on the bandwidth and processing power available (Section V-C).

In both cases, P_{mr} is executed repeatedly to establish whether adaptation (e.g., data reduction) is required to account for a network service degradation or whether adaptation is beneficial to prevent network under-utilization. To do so the application-level quality must be mapped down to network-level quality parameters such as the bandwidth required or the amount of data remaining to be shipped (d_{left}). d_{left}, together with the feedback on the available bandwidth, can be used to compute the time needed (t_{needed}) for the transfer. Corrective action must be taken if t_{needed} and the time left (t_{left}) differ "significantly". (Significance depends also on the size of the objects as well as network and application properties.)

B.1 Application-to-network QoS-mapping

The kind of QoS-mapping that enables the comparison between t_{needed} and t_{left} requires the application to provide a function $data(quality)$ that computes the amount of data necessary for a given object quality (see member function $Quality::data()$ in Fig. 4). d_{left} is then determined by the sum of $data(quality(obj))$ of the objects obj not yet delivered.

Given d_{left} and $bw(t) = get\_bw()$, which estimates the bandwidth available at time t in the future, we can compute t_{needed} by integrating (i.e. by summing up piecewise continuous parts of) the function $bw(t)$ over time t until an area (i.e. data volume) is covered which exceeds d_{left}. Thus, t_{needed} represents the time needed to transfer d_{left} given bandwidth $bw(t)$. This fairly general statement must be qualified to avoid misinterpretations: with a best-effort network service model $bw(t)$ can hardly be predicted for more than a few round-trip times with a reasonably high probability at the transport level (Section IV). Therefore, P_{mr} approximates the available bandwidth after these first few round-trip times with simple constant or linear functions based on past measurements. This approximation simplifies the computation of t_{needed}; knowledge about the bottleneck bandwidth is used to bound the approximation.

Note that it is only for the time needed to prepare and transmit the next object that P_{mr} needs to estimate future network behavior to be able to satisfy the user's request within the time limit—the reason is that the control loop gets an opportunity to take corrective action during the next iteration of P_{mr}, if required. In case we do not have such estimates, or if the conditions above cannot be met, e.g., because we are dealing with *large* objects, for which transmission takes longer than the system can reliably predict $bw(t)$, the situation is more complicated. Either the control loop gets a chance to take corrective action (because the time limit did not expire), or the data cannot be sent in the allot-

TABLE I
ABBREVIATIONS USED IN THE PAPER

P_{mr}	monitor- and react-phase
P_{prep}	prepare phase
P_{trans}	transmit phase
T	user specified time limit for response delivery
t_{left}	time left to deliver response, initialized to T
c_{prep}	CPU resources used to prepare objects for transmission
t_{prep}	time needed to prepare objects (given c_{prep}, $load(t)$)
t_{trans}	time needed to transmit objects (given $bw(t)$)
t_{needed}	time needed to deliver response (given t_{trans}, t_{prep})
t_{diff}	error variable of control loop ($t_{needed} - t_{left}$)
d_{left}	data remaining to be transmitted
$d_{reduction}$	reduction potential of an object
$bw(t)$	bandwidth estimation/prediction, $t > now$
$load(t)$	system load estimation/prediction, $t > now$

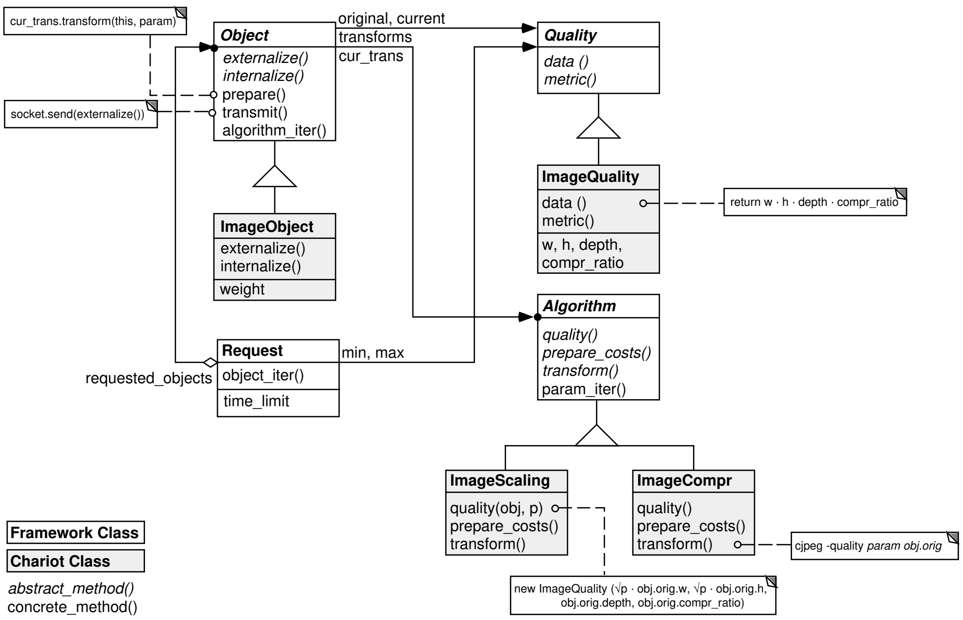


Fig. 4. Application-specific part of the class hierarchy (OMT-notation [33])

ted time. In the latter case, the application must be able to deal with the breakdown of the service model (Section V-E).

B.2 Network-to-application QoS-mapping

The goal of P_{mr} is to bring t_{needed} in line with t_{left} by either reducing or increasing the (overall) quality of the objects remaining to be delivered; these actions thereby reduce or increase d_{left}. The following questions must be considered while the sender tries to compensate for the difference $t_{diff} = t_{left} - t_{needed}$ by adapting the quality of the data awaiting delivery:

(i) Which object(s) should be chosen for adaptation (victim choice)?

(ii) How should the amount of quality adaptation be distributed among the chosen objects? Most importantly, how does the sender find the amount of quality adaptation needed given the volume of data adaptation required (d_{diff}) (quality distribution)?

(iii) Which algorithms should be used to accomplish a desired adaptation (algorithm selection)?

To a certain extent most of the questions above are application-specific and therefore cannot be answered in general. Thus, a framework for network-aware applications must provide flexibility in replacing, refining, or extending strategies as described in the following paragraphs.

(i) Victim choice: One strategy to chose objects for adaptation (i.e. victims) proceeds along the following idea: if quality reduction is required, choose the objects with the lowest weight-quality product, because they influence the overall quality the least. In case expansion is needed, the objects with the highest weight-quality product should be chosen for analogous reasons.

Note that the metric used for the victim choice depends on the application, e.g., in an image retrieval system it may be better to only decide according to the weights, which are based on similarity measures, because they reflect which images the user is really interested in.

(ii) Quality distribution: Given a set of victims to be reduced (or expanded), ideally the individual objects are reduced in quality inversely proportional to their weight-quality product (or their weight respectively). However, a problem arises

here because the system must satisfy two objectives at two different levels: On one hand, it aims to balance the application-level quality reduction according to the relative importance of the objects (i.e. their weight) and on the other hand, it needs to achieve a data reduction of a certain amount (d_{diff}) at the network level. The problem is that the mapping from network to application quality measures is generally ambiguous (in contrast to application-to-network mapping).

Example (Chariot): to effect an image size reduction of a factor N the server may either scale the image down by $\sqrt{N}$, reduce the color depth by a factor of N, find a JPEG quality factor that achieves such a compression ratio or use any combination of the image transformation algorithms mentioned.

Although we find transformations that achieve a certain data reduction, the direct effect on the quality reduction is not known since the application-level quality depends on the user-specified weighting of the individual quality attributes (e.g., resolution, color depth). This ambiguity makes it hard to guarantee balanced quality reduction and to find the required quality reduction efficiently in general. A straightforward but inefficient solution simply computes and compares the data and quality reduction of all the algorithms. Unless the application provides additional hints such as the continuity of the $data(quality)$ function, there is not much chance to improve upon such an approach. Finding efficient and generally applicable approaches to this aspect of application-controlled QoS-mapping is still an area of ongoing research. For our prototype system we make the simplification that $data(quality)$ is a linear function of $quality$; this assumption implies that the system must find only a "fair" distribution of d_{diff} that respects the weights of the individual objects (see Section V-D.2).

The problem of network-to-application QoS-mapping is further complicated since (a) the adaptation potential of an object (limited by the boundary conditions on min/max quality) must be taken into account, and (b) the transformations applied on the objects consume host resources and time. Therefore, the transformations indirectly impact t_{needed}. We address the issues related to (b) in Section V-C.

(iii) Algorithm selection: The choice of the (transformation) algorithm to accomplish a given quality adaptation is closely related to the issue of how much quality adaptation is required for each victim. There is usually an application-dependent choice as indicated in the example above. In our prototype framework we require the application to specify a list of transformation algorithms for each class of objects that can be part of a request. Each algorithm must provide a list of parameter values applicable. In addition, the application must provide functions that help the adaptation process estimate the data and quality reduction potential of an algorithm on a per-object basis.

C. System-awareness

Quality adaptations (e.g., by means of transformations, such as compression) cost CPU-resources and take a non-negligible amount of time to be completed. On one hand, a reduction in object quality may result in the desired reduction of transmission time, on the other hand, the transformations necessarily imply higher CPU-costs than simply retrieving an object (or image) from disk. Obviously, we want to avoid situations where a reduction of object quality in an attempt to reduce the error variable $|t_{diff}|$ to 0 incurs prepare costs (t_{prep}) that are higher than the gain in transmission time (i.e. $t_{prep} > t_{diff}$).

Therefore, our resource model also includes t_{prep}, the time needed for the phases P_{prep}. The adaptation process is still driven by network resource availability but additionally controlled by host resource consumption and availability. Each transformation algorithm registered for the requested objects must provide a function $prepare\_costs(obj, param, cpu)$ returning an estimate for the costs (c_{prep}) to transform obj from its original quality state to the one currently assigned on a given cpu. c_{prep} denotes the costs in terms of resources used, e.g., as given by *system* and *user CPU time* on Unix systems. c_{prep} is used to compute an estimate of the effective t_{prep} needed for a transformation by using an operating system dependent function $prepare\_time(c_{prep}, load)$, where $load$ denotes the average length of the process run-queue for example. (For most Unix systems the time needed for a given task using c_{prep} CPU time at a system load of $load$ can be approximated by $c_{prep} \cdot load$, up to a certain maximum load-level).

The effectiveness of the adaptation process and the reliability of the server to meet the QoS-constraints depend on the accuracy of all the models and estimates introduced in the last sections: $bw(t)$, $data(quality)$, $prepare\_costs(obj, param, cpu)$, $prepare\_time(cost, load)$, etc.. The more accurate the estimates used in the decision-making, the higher the probability that the sender is able to meet the time constraints.

Example (Chariot): The server computes c_{prep} as a function of image size and *param* used for the transformation algorithm. In contrast to approaches typically found in real-time systems, which rely on worst-case predictions for c_{prep}, our server bases its estimates on statistical data gained during past measurements of request processing. We derived regression models for both an algorithm's cost and its reduction potential. The regression models are regularly updated with new measurements.

C.1 Practical considerations: communication latency hiding

In a simple implementation of the software control loop, the phases of the framework execute sequentially. The adaptation produces stable results if $t_{prep} + t_{trans}$ for the adapted object is smaller than t_{trans} of the original object. However, sequential operation wastes bandwidth while the host is busy preparing the next object for transmission and wastes CPU resources while transmitting objects over a slow end-to-end path. With a slow connection, the sender is almost constantly congestion-controlled, and there are ample CPU cycles. An improved control loop tries to keep P_{trans} constantly sending and uses threaded prepare and transmit phases to hide the latency of the object delivery. *Communication latency hiding* calls for a different cost model: t_{needed} is no longer computed as $t_{prep} + t_{trans}$, but is approximated with $f \cdot t_{prep} + max(t_{trans}, (1-f) \cdot t_{prep})$, where f denotes the fraction of t_{prep} that is not available for latency hiding [38].

Although the intrinsics of the various resource models are outside the scope of this paper, the discussion above emphasizes the need for suitable abstractions. To allow for future refinements and extensions we encapsulate the computation and communication model deployed by a function $overall\_time(t_{prep}, t_{trans})$ that can be used to compute t_{needed}.

D. Using the framework to implement an adaptive system

Fig. 5 summarizes the steps involved in computing the error variable t_{diff} that drives the adaptation process. The function $compute\_t_{diff}()$ takes the request, i.e. the list of objects not yet transmitted, and the functions $bw(t)$, $load(t)$ as arguments. In addition, it uses the global variables t_{left} and cpu. $compute\_t_{diff}()$ is then used by the function $adapt()$, which is sketched in Fig. 6, and is invoked repeatedly by P_{mr} after obtaining new bandwidth feedback $bw(t)$. If t_{diff} exceeds an application-specific threshold ε limiting oscillation, the remaining objects in the *request* are subject to the adaptation process described in the next sections.

To accomplish the adaptation, the sender must find objects to transform. Given the list of objects that must be transmitted, there are several possible approaches to identify the victims, distribute the quality reduction, and select the transformation algorithms. We discuss here two such approaches.

D.1 General approach: exhaustive search

To avoid congestion and network under-utilization the adaptation process should aim to find a combination of objects to adapt and transformations to apply such that $|t_{diff}|$ is minimized and the overall quality metric is maximized. Unfortunately, an exhaustive search for the global minimum of $|t_{diff}|$ in the whole solution space is not attractive, as we illustrate in the next paragraphs.

Given a request consisting of N objects and given M transformation algorithms each taking m different parameter values on average, there are $n \approx \sum_{i=0}^{M} \binom{M}{i} m^i$ possible transformations applicable to each of the objects. If we assume that all the possible combinations fulfill the QoS-restrictions, there are approximately N^n possibilities to adapt the request to the currently available bandwidth. In each iteration of P_{mr}, the sender must compute t_{diff} for each of the N^n points in the solution space and, e.g., find the combination with the smallest $|t_{diff}|$. As an alternative, the sender can try to find the combinations with $|t_{diff}| < \varepsilon$ and choose the one with maximal $\sum weight \cdot quality$.

As long as there is no additional information about the functions used to compute t_{diff} (e.g., gradients), or as long as the quality boundaries are not very restrictive, the size of the solution space cannot be reduced, and hence the complexity is too high to make this approach feasible in the general case. Therefore, we cannot include a generic method to perform exhaustive search in the framework, since we expect the methods of the framework to provide a solution for all possible extensions. However, we can provide the application with several strategies [14] for the adaptation process (one being exhaustive search for example) and leave it to the application developer to decide on the most appropriate strategy to use in the context of the application.

D.2 A practical approach: approximative search

If N or n are large, the sender must either employ some approximations or introduce simplifications in the adaptation process to reduce the complexity of the adaptation process, otherwise the search is so expensive that the resource consumption of P_{mr} must be included in the cost models. For the sake of simplicity we restrict our discussion to the former case.

The idea that forms the basis of the currently implemented adaptation process is to approximate the search for a minimal $|t_{diff}|$ by iteratively trying to apply the possible transformation algorithms with their respective parameters with the objective to find a local minimum that is within the tolerance. If one algorithm does not achieve the desired result, the next algorithm is chosen [24]. The adaptation phase, i.e. the $reduce()$ function in Fig. 6, then proceeds along the following steps (see Section V-B.2):

(i) the victims are chosen as the first n objects from the request list, which is ordered by increasing weights, such that $\sum_{i=0}^{n} d_{reduction}(obj_i) \geq d_{diff}$. d_{diff} represents the amount of data reduction that is required to compensate for t_{diff}. $d_{reduction}(obj)$ denotes the reduction potential of the current quality state of object obj, which is bounded by the minimal quality tolerated by the user. If no such set of n objects exists with which the necessary data reduction can be achieved an exception is thrown, which is caught and handled in $adapt()$ (Fig. 6).

(ii) With the simplifying assumption $data \sim quality$, d_{diff} is distributed among the victims by assigning the reduction needed for each object to a fraction of d_{diff} inversely proportional to the object's relative weight (in the request list), unless $d_{reduction}$ poses a limit on the reduction attainable. In such a case, the distribution step is repeated, as long as there are objects whose reduction is limited by $d_{reduction}$ and as long as t_{diff} has not been fully compensated for.

(iii) The transformation algorithm selection is done by iterating over the algorithms, the objects, and the parameters. In each step t_{diff} is computed and the iteration terminates when $|t_{diff}| < \varepsilon$.

Note that the adaptation process outlined makes heavy use of the iterators shown in Fig. 4. Use of iterators facilitates experimentation with different priorities of the transformation algorithms used. Based on our experience with Chariot, we found that being able to cleverly apply application-knowledge to set priorities is essential for the effectiveness of the approximative search.

E. Problems with feedback control

This paper describes the overall structure of a framework for network-aware applications. Several practical issues have not been mentioned or discussed in detail:

Start-up behavior: Special care must be applied to find the optimal operating point of the control loop as soon as possible while avoiding overshooting and an excessively conservative (i.e. slow) start-up. For a network-aware sender this requirement means that the server ought to start delivering objects as soon as possible to get early feedback. Furthermore, the sender should refrain from sending too large an object at the start, in case bandwidth turns out to be unexpectedly low. These requirements impact application design as follows: an application should either (i) allow the list of requested objects to be reordered, such that objects obj with small $data(obj)$ that need not or cannot be adapted are sent first; (ii) be able to cope with an interrupted object delivery that may be restarted in lower

$$
\begin{aligned}
d_{left} &= \sum_{obj \in request} data(quality(obj)) \\
t_{trans} &= transmit\_time(d_{left}, bw(t)) \\
c_{prep} &= \sum_{obj \in request} prepare\_costs(obj, algorithm(obj), param(obj), cpu) \\
t_{prep} &= prepare\_time(c_{prep}, load(t)) \\
t_{needed} &= overall\_time(t_{prep}, t_{trans}) \\
t_{diff} &= t_{needed} - t_{left}
\end{aligned}
$$

Fig. 5. Function $compute\_t_{diff}(request, bw(t), load(t))$ returning t_{diff}

```
t_diff = compute_t_diff(request, bw(t), load(t))
try {
  if ( t_diff > ε )                // prevent congestion
    t_diff = reduce(request, bw(t), load(t))
  else if ( t_diff < -ε )          // prevent under-utilization
    t_diff = expand(request, bw(t), load(t))
  assert ( |t_diff| ≤ ε );
} catch (NoAdaptationPossible exception) {
  handle (exception);              // application specific handler
}
```

Fig. 6. Function $adapt(request, bw(t), load(t))$

quality; or (iii) support hierarchical encoding and progressive delivery of objects, such that the transmission can be stopped at any time.

Bandwidth probing by the lower layers of the communication system allows to estimate the expected bandwidth after just a few RTTs (e.g., packet-pair probing [20], [30]) and can also help to alleviate the problems with start-up behavior.

Communication idle time: Gaps in the sequence of object transmissions should not only be avoided because of the transmission opportunities lost at the application level, but also because many congestion control mechanisms exhibit a *use-it-or-lose-it* property [11]. That is, communication idle time results in loss of the fair share of the bottleneck bandwidth previously held by the connection and consequently results in repeated start-up behavior.

Latency of prepare and transmit activities: With our model of dynamic adaptation to network service quality, a network-aware sender must rely on either good bandwidth estimates or on the expectation that network service does not degrade more during $t_{prep} + t_{trans}$ of the next object than there is data reduction potential inherent to the remaining objects in the request list. Due to the nature of best-effort network service these assumptions may not be fulfilled. Such a situation results in the breakdown of the service model.

Ill-specified boundary conditions are another cause of failure that requires application-specific reaction. No application should set T and then require a high minimal quality such that even sending at minimal quality exceeds the time limit. However, the appropriate settings of the boundary conditions cannot always be anticipated. Therefore, an application must be able to deal with such situations. Possible reactions include delivery of objects at minimal quality (desirable in an image retrieval system), a user-application dialogue to renegotiate the boundary parameters, or termination of transfers altogether. This last option is attractive if it allows an overloaded server to catch up. The application-provided exception handler in Fig. 6 deals with such situations.

VI. Evaluation

This section presents results from the Chariot system, which is an extension of the framework presented here. We concentrate here on assessing the ability of the (adaptive) server to respond to bandwidth fluctuations, i.e. its network-awareness. Note that the examples presented here serve the purposes of validating the approach as well as pointing out areas of further research. The restricted nature of selected examples can by no means replace an extensive evaluation and quantification of the adaptation potential in practice. However, such a study is beyond the scope of this paper.

A. Evaluation Methodology

Our approach to evaluate the system's network-awareness proceeds in two steps: First, we subject the system to synthetic reference bandwidth waveforms (the example presented here is the *Step-Down* waveform shown in Fig. 7a) to characterize its ability to adapt in general and in accordance with the (well-established) principles for measuring dynamic response from the field of *control systems* [32]. Second, field tests in the Internet with its high bandwidth dynamics enable us to assess the system's agility with respect to real-world network traffic.

Since ensuring reliable and reproducible experiments on real networks is extremely difficult, we follow the approach of other researchers and resort to a technique called *trace modulation* [28]. Trace modulation performs an application-transparent emulation of a slower target network on a faster, wired LAN. Each application's network traffic is delayed according to delay and bandwidth parameters read from a so-called replay trace, which is gathered from monitored transfers.

B. Experimental Setup

In our experiments the Chariot server runs on a 150 MHz MIPS R4400 SGI Challenge S with 128 MB of memory. A 134 MHz MIPS R4600 SGI Indy with 64MB of memory serves as the platform for the client. For both of the experiments shown below, the client requests transmission of 90 JPEG images stored at the server in a resolution of 380×250 pixels and a JPEG quality factor of 95–97. The 90 images total 5.2 MB of data to be transmitted. The images are assumed to be equally relevant, which means that equal weights are assigned to the 90 images. The user-imposed time limit for request processing is arbitrarily chosen to be 60 seconds with a tolerance interval of [–2, 2] seconds.

The bandwidth replay traces used for the two experiments conducted are depicted in Fig. 7. The *Step-Down* waveform of Fig. 7a is an idealization of real network scenarios; it approximates possible situations in an overlay network for instance, where a mobile client may seamlessly switch between different network interfaces. Fig. 7b shows the monitor layer's perception of the available bandwidth during a transfer between the ETH Zürich (Switzerland) and the University of Linz (Austria). This bandwidth curve has been smoothed using a two second averaging interval. Hence the system under test does not deal with the problems of start-up behavior.

The Chariot server operates using the "approximative search" adaptation process described in Section V-D.2. Chariot's reduction algorithms registered with the framework are image scaling (with factors 1/2 and 1/4) and image compression (with quality factors 75, 50, and 25 [18]). The server performs communication latency hiding by means of a separate thread for P_{prep}. As a consequence, P_{trans} for image i of the sequentially processed request list operates concurrently to P_{prep} for image $i+1$.

C. Experimental Results

C.1 Step-Down waveform

Fig. 8—a data vs. time plot as introduced in [19]—shows that Chariot is able to both adapt the amount of data transmitted (curve named "actual") to the amount of data transmittable ("possible") and deliver the 90 images within the 60-second time limit. The *Step-Down* waveform of the available bandwidth in Fig. 7a represents the derivative of the curve named "possible". The sharp drop in bandwidth at $t = 30$ seconds is absorbed almost without loss of transmission possibilities. Loss of transmission possibilities, which is characterized by the vertical difference between the curve showing the data theoretically transmittable ("possible") and the data actually transmitted ("actual"), can be caused by prepare or control loop overhead. The curve depicting the control loop's estimate of the total amount of data transmittable within the time limit ("estimated") shows that the adaptation at $t = 30$ took place swiftly (within a small fraction of a second). The estimate is based on the amount of data already transmitted, the monitor's estimate of the available bandwidth $bw(t)$ and t_{left}.

Fig. 9 plots the control loop's error variable t_{diff} that drives Chariot's adaptation. The two horizontal lines at $t_{diff} = 2$ s and $t_{diff} = -2$ s represent the tolerance interval specified. The "time difference" plot shows that in fact three different (major) adaptation events occurred (adaptation is necessary when $|t_{diff}| > 2$ s). First, around $t = 0$ s initial adaptation steps are necessary to reduce the 5.2 MB to the 4.7 MB estimated to be transferable. Second, due to the sharp bandwidth drop at $t = 30$ s, t_{needed} and hence t_{diff} increase by approx. 33 seconds; this drop is compensated in subsequent reduction steps. Third, t_{diff} exceeds the 2 second-tolerance twice at $t \approx 33$ s although no change in bandwidth could be observed. This fact may be attributed to inaccuracies in the estimates of c_{prep} and the reduction potential of images. Although provision of inaccurate estimates by the application can have a detrimental impact on the overall performance (i.e. the quality deliverable), the example shows that our control loop mechanism is flexible enough to even cope with such situations.

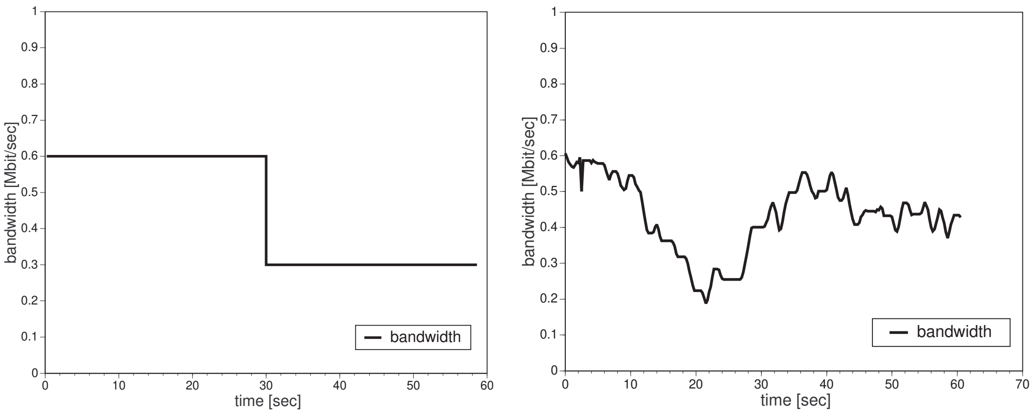
C.2 Internet traffic

Fig. 10 shows that Chariot is even capable of dealing with frequent oscillations in the available bandwidth as present on today's wide-area network paths. Note, however, that the penalty in terms of transmission possibilities lost is higher than in the previous case. The curve depicting the data volume transmittable ("possible") relates to the bandwidth waveform shown in Fig. 7b. Careful examination of the curve plotting the data effectively transmitted reveals two cases (at $t \approx 3$ s and $t \approx 20$ s) where transmission lulls had to be accepted. The reason is that in these cases P_{trans} for image i finished before the concurrently executed phase P_{prep} for image $i+1$ and thus had to wait before starting transmission of image $i+1$. The causes for this behavior can be twofold: Either $c_{prep}(img_{i+1}) > t_{trans}(img_i)$, in which case the adaptation process could try to reorder the images in the request list to avoid communication idle time, or the server's load is too high, such that $t_{prep}(img_{i+1}) = load \cdot c_{prep}(img_{i+1}) > t_{trans}(img_i)$. The latter problem calls for host resource reservation by the operating system as other researchers have suggested [25], [26].

Keep in mind, that although the examples presented show that adaptation to meet the given time limit works, the whole process of adaptation is quite sensitive to the choice of the "boundary conditions", such as the time limit. Since the adaptation potential is limited by the reduction potential of the objects/images to be transmitted and the cost incurred for their transformation, unrealistic expectations from the user may simply result in the break-down of the service model.

VII. Related Work

We can divide approaches to provide predictability of service quality to the application/user into two categories: those that are



(a) Step-Down waveform

(b) Bandwidth of Internet image transfer

Fig. 7. Bandwidth replay traces used

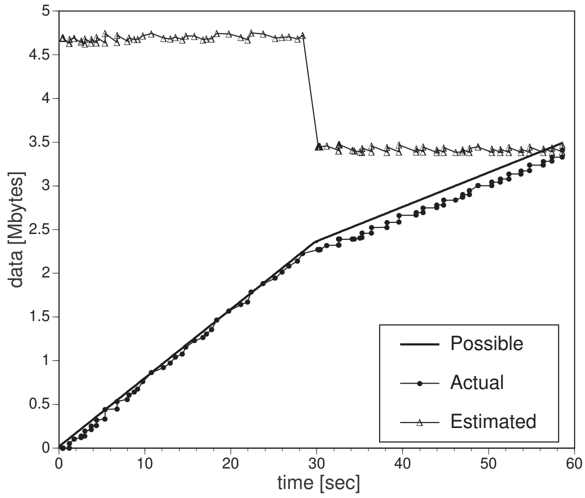


Fig. 8. Data volume transmitted in *Step-Down* scenario

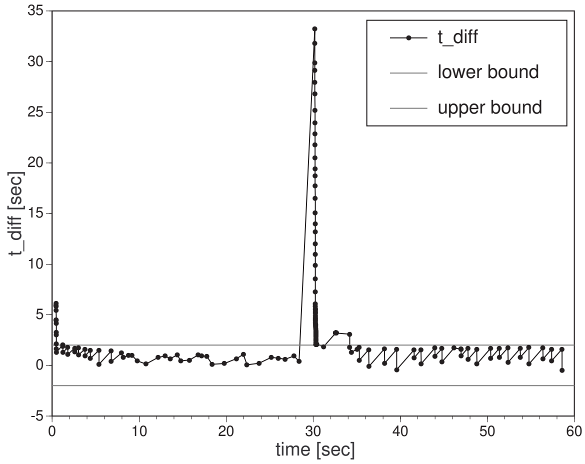


Fig. 9. Time difference (t_{diff}) plot for *Step-Down* example

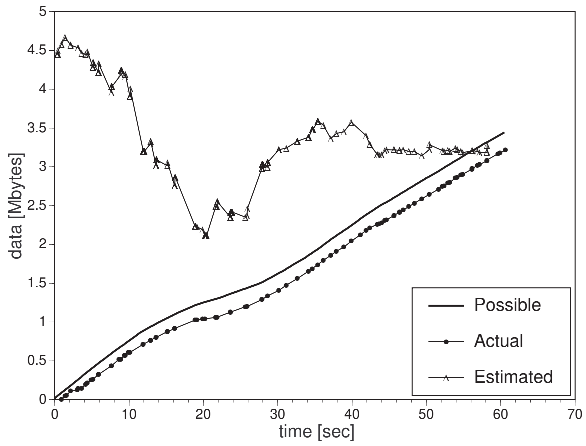


Fig. 10. Data volume transmitted from Zürich to Linz

based on reservations and those that are based on adaptation (see Section II-A).

A. Reservation

There exists a long tradition of research into reservation of network resources, with a trend towards integrating multiple service models in a single cell- or packet-switched network [8], [40]. It has been recognized that to support end-to-end QoS guarantees not only network aspects must be considered, but the end-system and OS-resources must also be taken into account [26]. This requirement holds especially for continuous media applications as they have the most stringent resource requirements [36], [34]. In step with advances in resource guarantee provision in both fields, researchers identified the need for resource orchestration and developed methods that allow for meeting the user's QoS requirements on an end-to-end basis [25], [6]. Most methods involve QoS-negotiation procedures mainly based on application-to-network QoS-mapping.

B. Adaptation

Adaptation is an effective way of enhancing the user's perception of service quality in environments where resource reservation is not possible, or in situations where it is impossible for an application to specify its resource requirements in advance.

Recent adaptive system's such as RLM [22], [23] or IVS [2] have shown that even continuous media applications can benefit from adaptation in environments lacking reservation capabilities. Their feedback-driven adaptation scales back quality and hence resource consumption when application performance is poor, and they attempt to discover additional resources by optimistically scaling up usage from time to time. While IVS employs sender-based bandwidth adaptation, RLM pioneered receiver-based adaptation in a multicast environment. Also, both systems continuously adapt their *play-out point* to account for variations in the transmission latency.

In contrast to these systems, Odysee [27] seeks to provide a more general approach to the construction of resource-aware applications by modifying the interface between applications and the operating system. Their measurement-based approach employs receiver-driven adaptation and concentrates on orchestrating multiple concurrent resource-aware applications on the client rather than on the server. In contrast, our framework uses sender-based adaptation and identifies a wide range of methods that can be customized by the user.

Fox et al. [13] propose a proxy-based architecture employing so-called *distillation services* to adapt the quality of the service for the client to the variations in network resource availability. Their system—in addition to being network-aware—also accounts for variability in client software and hardware sophistication.

VIII. Concluding remarks

This paper presents a simple framework for the construction of network-aware applications. Given the framework, the application developer must specify functions to determine the relationships between quality and size as well as provide estimates on the effectiveness of various transformations to reduce size. Fig. 5 summarizes the functions required.

Undoubtedly, further work is required to find more elaborate solutions to the problems discussed in this paper. However, the abstractions identified in the adaptation process allow for experimentation with various methods for information collection and with methods providing better estimates, such that tradeoffs can be found between the accuracy achieved, the efforts involved in providing the estimates, and their effect on the bandwidth adaptation. As it is not always possible to provide good estimates for network behavior (or for the application's resource demands) it is important that systems are designed for adaptivity. Such systems can observe the actions involved with a decision and can take corrective action if necessary. A framework provides the context for such experimentation by application developers, and frees the developer from the need to acquire a detailed understanding of the monitoring system, network protocols, the network interface or router capabilities. Our experience with the development of an adaptive image server has demonstrated the practicability and benefits of this approach.

The development of network-aware application requires considerable effort, and no amount of adaptation can accomplish the impossible—satisfy unrealistic expectations by an application or a user. However, with adaptation, applications can push the envelope of acceptable network performance, and we expect increased use of adaptation techniques both in stationary and mobile network applications. The framework outlined here provides an approach that shelters the application developers from many details of adaptivity and thus helps to reduce the effort involved in the development of network-aware applications.

ACKNOWLEDGEMENTS

We thank S. Blott, P. Brandt, A. Dimai, R. Karrer, M. Näf, M. Stricker, P. Walther and R. Weber for their contributions to the design and implementation of the Chariot system. We appreciate the feedback of the referees which improved the paper considerably. Finally, we acknowledge the discussions during the workshop on network-aware and mobile applications held in conjunction with ESEC/FSE '97 in Zürich.

REFERENCES

[1] J. Bolliger, P. Brandt, A. Dimai, T. Gross, R. Karrer, and R. Weber. Architecture of a networked image search and retrieval system. Report, ETH Zurich (Dept. of Computer Science and Dept. of Electrical) Engineering, 1997.

[2] J-C. Bolot, T. Turletti, and I. Wakeman. Scalable feedback control for multicast video distribution in the Internet. In *Proc. ACM SIGCOMM '94*, pages 58–67, London, August 1994.

[3] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: New techniques for congestion detection and avoidance. In *Proc. ACM SIGCOMM '94*, pages 24–35, London, October 1994.

[4] D. Brennwalder. Adaptives Transportprotokoll (in German). Master's thesis, Dept. of Computer Science, ETH Zürich, March 1997.

[5] R. Cáceres, P. B. Danzig, S. Jamin, and D. J. Mitzel. Characteristics of wide-area TCP/IP conversations. In *Proc. ACM SIGCOMM '91*, pages 101–112, Zürich, August 1991.

[6] A. Campbell, G. Coulson, F. Garcia, and D. Hutchison. A continuous media transport and orchestration service. In *Proc. ACM SIGCOMM '92*, pages 99–110, Baltimore, Maryland, August 1992.

[7] T. M. Chen, S. S. Liu, and V. K. Samalam. The available bit rate service for data in ATM networks. *IEEE Communications Magazine*, 34(5):60–71, May 1996.

[8] D. D. Clark, S. Shenker, and L. Zhang. Supporting real-time applications in an integrated services packet network: Architecture and mechansim. In *Proc. ACM SIGCOMM '92*, pages 14–26, Baltimore, Maryland, September 1992.

[9] H. Eberle. A workstation interconnect supporting time-dependent data transmission. In *Multimedia'94*, pages 7.1.1–7.1.6, Kyoto, May 1994. IEEE Communication Society.

[10] H. Eberle, and E. Oertli. Switcherland: A QoS communication architecture for workstation clusters. to appear in *Proc. ACM ISCA '98*, Barcelona, Spain, June 1998.

[11] K. W. Fendick. Evolution of controls for the available bit rate service. *IEEE Communications Magazine*, 34(11):35–39, November 1996.

[12] S. Floyd and K. Fall. Router mechanisms to support end-to-end congestion control. Technical report, Lawrence Berkeley National Laboratory, Berkeley, February 1997.

[13] A. Fox, S. D. Gribble, E. A. Brewer, and E. Amir. Adapting to network and client variability via on-demand dynamic distillation. In *ASPLOS-VII Proc., 7th Intl Conference on Architectural Support for Programming Languages and Operating Systems*, pages 160–173, Cambridge, MA, October 1996.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Object-Oriented Software*. Addison Wesley, 1995.

[15] M. Hayter and D. McAuley. The desk-area network. *ACM Operating Systems Review*, 25(4):14–21, October 1991.

[16] J. C. Hoe. Improving the start-up behavior of a congestion control scheme for TCP. In *Proc. ACM SIGCOMM '96*, pages 270–280, Stanford, CA, August 1996.

[17] IEEE. *Software Engineering Standards (3rd Ed.)*. IEEE Press, New York, 1989.

[18] Independent JPEG Group. *IJG's JPEG Software Release 6a*. Independent JPEG Group, February 1996.

[19] V. Jacobson. Congestion avoidance and control. In *Proc. ACM SIGCOMM '88*, pages 314–329, August 1988.

[20] S. Keshav. A control-theoretic approach to flow control. In *Proc. ACM SIGCOMM '91*, pages 3–15, Zürich, August 1991.

[21] M. Mathis and J. Mahdavi. Forward acknowledgement: Refining TCP congestion control. In *Proc. ACM SIGCOMM '96*, pages 281–292, Stanford, CA, August 1996.

[22] S. McCanne and V. Jacobson. vic: A flexible framework for packet video. In *Proc. ACM Multimedia '95*, pages 511–522, San Franscisco, CA, November 1995.

[23] S. McCanne, V. Jacobson, and M. Vetterli. Receiver-driven layered multicast. In *Proc. ACM SIGCOMM '96*, pages 117–130, August 1996.

[24] M. Näf. Ein adaptives Bildtransferprotokoll fuer Chariot (in German). Internal report, Dept. Computer Science, ETH Zürich, March 1997.

[25] K. Nahrstedt and J. Smith. The QoS broker. *IEEE Multimedia*, 2(1):53–67, Spring 1995.

[26] K. Nahrstedt and R. Steinmetz. Resource management in networked multimedia systems. *IEEE Computer*, 28(5):52–63, May 1995.

[27] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Agile application-aware adaptation for mobility. In *Proc. 16th ACM Symposium on Operating System Principles*, October 1997.

[28] B.D. Noble, M. Satyanarayanan, G. T. Nguyen, and R. H. Katz. Trace-based mobile network emulation. In *Proc. ACM SIGCOMM '97*, pages 51–62, Cannes, France, September 1997.

[29] V. Paxson. Empirically-derived analytic models of wide-area TCP connections. *IEEE/ACM Trans. on Networking*, 2(4):316–336, August 1994.

[30] V. Paxson. *Measurements and Analysis of End-to-End Internet Dynamics*. PhD thesis, University of California, Berkeley, April 1997.

[31] K. K. Ramakrishnan and R. Jain. A binary feedback scheme for congestion avoidance in computer networks with a connectionless network layer. In *Proc. ACM SIGCOMM '88*, pages 303–313, August 1988.

[32] O. Rubin. *The Design of Automatic Control Systems*. Artech House, Norwood, MA, 1986.

[33] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[34] H. Schulzrinne. Operating system issues for continuous media. *Multimedia Systems*, 4:269–280, 1996.

[35] H. Schulzrinne, S. L. Casner, R. Frederick, and V. Jacobson. RFC 1889: RTP: A transport protocol for real-time applications. Request for Comments, January 1996.

[36] R. Steinmetz. Analyzing the multimedia operating system. *IEEE Multimedia*, 2(1):68–84, Spring 1995.

[37] R. W. Stevens. *UNIX Network Programming*. Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[38] V. Strumpen. *The Network Machine*. PhD thesis, Dept. of Computer Science, ETH Zürich, July 1995. No. 11227.

[39] Taligent. Inc. Building object-oriented frameworks. 1994.

[40] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A new resource reservation protocol. *IEEE Network*, 7(5):8–18, September 1993.

[41] J. Zinky, D. Bakken, and R. Schantz. Architectural support for quality of service for CORBA objects. *Theory and Practice of Object Systems*, 3(1):55–73, 1997.

Jürg Bolliger received the dipl. Informatik Ing. degree (computer science) in 1994 from the Swiss Federal Institute of Technology (ETH), Zürich, Switzerland. In 1994, he was with the Symbolic Computation Group at the University of Waterloo, Canada. Since 1995, he is a Ph.D. student in the Department of Computer Science at ETH Zürich. He is working on network-aware applications, their interaction with network protocols and on congestion control issues in transport protocols.

Thomas Gross received a Ph.D. from Stanford University (1983). He is interested in the design, implementation, and evaluation of software systems. He has participated in a number of research projects (including the MIPS project at Stanford, the Chariot project at ETH Zürich, and the Warp, iWarp, Fx, cmcc, and Remulac projects at Carnegie Mellon) and has taught classes on computer architecture, parallel computing, software construction, and programming techniques. He has been drafted to be a member of several program committees, and too often editors of journals try to press him into service as a reviewer. In his "free" time, he investigates *network-aware* computing.