A Perspective on Application/Network Coupling

Thomas Gross[1,2] and Peter Steenkiste[1]

[1]School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[2]Departement Informatik
ETH Zürich
CH 8092 Zürich

Abstract

Networked applications execute in a distributed environment that is shared with many users, so the level of service they receive from the network can be highly variable. Since the characteristics of the network often determine overall application performance, network-aware applications attempt to adjust to changes in the network conditions. Therefore, such applications must be able to obtain performance-related information about the network. We discuss two dimensions of the coupling of such applications to the network system. One dimension focuses on how the application learns about the state of the network: The application can use implicit information, e.g. the packet loss rate it observes, or it can receive explicit information from the network about topology and dynamic traffic conditions. The other dimension covers the model of interaction between application and network: Either the information is collected continously and the application controls when it acquires information about the network (e.g., the application consults a database maintained by the network), or the interaction is triggered by specific network events (e.g., the network notifies the application of significant changes). Network-aware applications today almost exclusively use the implicit style of information collection, in part because it does not require any support from the network. In this paper we argue that explicit information is needed as well if we want to give applications the opportunity to explore the full range of adaptation options. To demonstrate the benefits of explicit coupling, we developed Remos (Resource Monitoring System), an interface that allows applications to collect information about network conditions. We briefly describe the design and prototype implementation of Remos.

1 Introduction

Many networked applications are very sensitive to timely responses of the communication system. Integrated applications that mix traffic types (data, video, audio, control) often face difficulties to obtain adequate network services from today's networks. The response of network architects to such application demands has been the definition of different service classes. Some service classes provide hard guarantees (e.g., the IETF guaranteed service), or weaker "guarantees" (e.g., the IETF controlled load service). Researchers are also exploring the notion of service differentiation, i.e. multiple classes with different (relative) service quality are provided, but no guarantees are made about the (absolute) level of service that will be received.

This development is a mixed blessing for application developers. Guaranteed service is an attractive solution, but it is hard to use and likely to be relatively expensive, especially for bursty traffic, and it may not be deployed for a long time. Service classes such as controlled load are easier to deploy and use, but since they do not provide strict guarantees, applications will have to be able to adapt within some window of network performance. Best effort services are likely to be most widely available and most economical, but applications may have to adapt over a very wide performance window. The net result is that while some of the proposed service classes will make it easier to develop applications that transfer audio, video, and time sensitive data, most service classes still require applications to adapt to dynamic network conditions. Moreover, the application (developer) must now also select an appropriate service class.

2 Problem

It is advantageous for applications to be network aware since it allows them to run under a broader range of conditions, and it may also allow them to use less expensive service classes, e.g. best effort instead of guaranteed service. Some of the functionality (and complexity) associated with adaptation can be embedded in middleware, but we first have to understand the adaptation process before we develop widely usable libraries or middleware layers.

2.1 Space of feedback options

To adapt, applications need information on the status of the network. Traditionally, this task has been performed by communication protocols, such at TCP, so it is worthwhile to look at how these protocols collect information about network conditions. Protocols are often classified as using *implicit* or *explicit* feedback from the network. In protocols based on implicit feedback, the receiver monitors the incoming data stream and uses this stream to derive information about network conditions. TCP is good example: dropped packets are viewed as a sign of congestion, and the sender responds by reducing its rate. In contrast, with explicit feedback, some entity inside the network provides explicit information about network conditions to senders. A good example is the ATM ABR traffic class: senders receive periodic information about network congestion conditions (e.g., congestion bit in rate management cells), or even the specific maximum traffic rate they are allowed to use (e.g., EPRCA).

Implicit feedback has the advantage that it does not require support from the network so this approach to provide feedback is always feasible[1]. Implicit feedback has also disadvantages: (i) it only allows incremental adaptation (i.e., when two hosts communicate, implicit information provides updates on how the bandwidth between these hosts evolves), and (ii) it is sometimes difficult to interpret the "information". Explicit feedback is in general easier to use, but it requires network support. Protocols today primarily rely on implicit feedback, and the same is true for initial network-aware applications. The reason is simple: implicit feedback does not require networking support, which does not exist.

Explicit information can be provided to the application in two ways. First, the network can provide feedback continuously. This approach is, e.g., employed for ABR traffic: a rate management cell is exchanged with the network for every 32 data cells. Continous feedback is most often based on network properties that subsequently must be interpreted in the application space; for this reason this kind of coupling is also called *indirect*. An alternative is that applications receive a notification when specific events happen, e.g, an application receives an asynchronous notification when the network bandwidth drops below a certain threshold or when the connection switches from one type of network to another [7]. Such notifications can be in the form of callbacks, or by invoking a specific event handler. With this style of interaction, the relationship between the network event (e.g., drop in bandwidth) and the actions (by the application or protocol software) is clearly established (e.g., when registering the handler). Therefore we call this style *event-driven* or also direct coupling.

[1] Implicit feedback typically makes some assumptions about the network, e.g., it considers packet loss to be a sign of congestion.

Finally, it useful to distinguish between the interface used by the application and the functionality supported by the network, since it is possible for a library or middleware layer to translate one interface into another. For example, a library could translate continuous network feedback into event-based application feedback. A more interesting approach includes activities by the middleware: middleware could use a set of benchmarks to collect information on the conditions in a network (that does not provide explicit feedback) and present this information to the application in an explicit form. We focus on the application level interface, and we will touch on the lower level interface when we discuss implementation options.

2.2 Space of application adaptation options

The above discussion indicates that network-aware applications may need to adapt to the network in two ways: (i) pick the most appropriate service class, if multiple service classes are available, and request service (if not a best effort class); (ii) adapt to the service level available in the current service class. The first step requires an interface that allows applications to discover what services are available. The second step can in principle be done using implicit network feedback, e.g., reduce the frame rate or size if the receiver observes corrupted frames, and periodically probe for more bandwidth if the connection is lossless. This model of network-awareness is appropriate for many applications and can be encapsulated in a framework[1].

However, there is more to network awareness than adjusting frame rates for video streams or generating the appropriate parameters needed to create a connection with delay guarantees. The above model focuses on the "steady state" (of adaptation, not necessarily a steady state in the network!). E.g., it leaves open the issue of how to deal with start up, and it also does not support more interesting forms of adaptation, e.g., switching to a higher or lower quality service class during execution.

If we look at the general problem of how applications can adapt to network conditions, we see a multi-dimensional space of options. The three main dimensions are time, space, and service classes.

Along the *time* dimension, applications dynamically adapt to changing network conditions. Adaptation is in general based on the assumption that recent past conditions are a good predictor of near-term future conditions, an assumption that often holds. Adaptation along the time dimension is addressed by protocols such as TCP, and has also received the most attention from researchers studying network-aware applications.

Adaptation along the *space* dimension reflects the fact that applications sometimes have a choice about what nodes, and thus what part of the network, to use. A simple example is a Web browser that can choose which of several replicated servers to use, or that can select a proxy cache that may have a potentially stale copy of a video clip. Another example is a distributed simulation that has a choice of how many and what nodes it should use.

The *service class* dimension reflects the fact that the application has to pick a service class, and this decision may (should) be based on dynamic conditions. E.g., when setting up a video conference over a network that supports differentiated service, the user or application will want to pick the lowest service class, if that can provide sufficient bandwidth. A higher service class will be selected if it can deliver the bandwidth while the lower service class cannot.

2.3 Network-application interactions

To be able to adapt along all three dimensions, applications will need information on network conditions, which span a matching space with the same dimensions. How applications collect network information impacts how easily applications can explore this space.

Implicit information is based on experience, which severely restricts what part of the information space is explored: the application only learns about the part of the space it currently operates in. This means that it can only collect information on the network conditions along the paths it is currently using and on the service class it is in. Implicit feedback does provide information along the time dimension, but only while the application is actively using the network. At startup or after the application has been idle for a while, no useful information is available.

We argue that given the limits on what information can be collected using implicit feedback, mechanisms must be provided so that applications can get explicit information on network conditions, e.g., by querying an standard interface. Such an interface should allow applications to collect information on network conditions in the entire network space (space, time and service class dimensions), allowing applications to make adaptation decisions in space, across service classes, and at startup.

One can argue that the restrictions on the type of information that can be collected using implicit feedback is not fundamental. Applications can use probing to explore the entire information space, e.g., they can periodically try all service classes and they can measure measure the network performance between every pair of usable hosts. While this approach may be appropriate in some cases, it is in general undesirable. First, developing effective network probing routines is difficult; it is not something that application developers should be required to do. Second, probing can be expensive, both in terms of elapsed time for the application and consumed network resources. Furthermore, excessive probing may disturb the measurements taken by this and other applications. In fact, large scale probing by applications would negate many of the advantages of implicit feedback. If probing is needed, it should be performed by a middleware layer, which would mean that the code can be developed as part of the network architecture, and the collected information can be shared by many applications.

Note that we are proposing explicit feedback as a complementary mechanism to implicit feedback, and not as a replacement. The advantages of implicit feedback, if used appropriately, remain. Implicit feedback will remain useful as an inexpensive way of getting continuous feedback, once a particular operating point along the space and service dimensions has been picked. Implicit feedback is also like to give more accurate and timely information (in a narrower part of the operating space) than explicit feedback.

3 Example: Remos

The Remos API provides a query-based interface that allows clients to obtain “best-effort” information [3] on network conditions. The applications specifies the kind of information it needs, and Remos supplies the best available information. To limit the scope of the query, the application must select network parameters and parts of a larger network that are of interest. In this section we briefly describe the main Remos features. A more detailed description can be found elsewhere [3, 6].

3.1 Level of abstraction

To accommodate the diverse application needs, the Remos API provides two levels of abstraction: high level flow-based queries and lower level topology-based queries.

Remos supports flow-based queries. A *flow* is an application-level connection between a pair of computation nodes. Queries about bandwidth and latency of sets of flows form the core of the Remos interface. Using flows instead of physical links provides a a high level of abstraction that makes the interface portable and independent of system details. Flow-based queries place the burden of translating network-specific information into application-oriented information on the implementor of the API. However, flows are an intuitive abstraction for application developers and they allow the development of adaptive network applications that are independent of the heterogeneity inherent in a network computing environment.

Remos also supports queries about the network *topology*. The reason we expose a network-level view of connectivity is that certain types of questions are more easily or more efficiently answered based on topology information. For example, finding the pair of nodes with the highest bandwidth connectivity is expensive using only flow-based queries.

The topology information provided by Remos consists of a graph with compute nodes, network nodes, and links, each annotated with their physical characteristics, such as latency and available bandwidth. Topology queries return a *logical* interconnection topology. This means that the graph represents the network behavior as seen by the application, and does not necessarily reflect the physical topology. Using a logical topology gives Remos the option of hiding network features that do not affect the application. For example, subnets can be replaced by (logical) links if their internal structure does not affect applications. Topology information is in general harder to use than flow-based information, since the complexity of translating network-level data into application-level information is mostly left to the user.

3.2 Dynamic resource sharing

Since networks are a shared resources, it is important to account for how resources are shared by multiple flows. Since multi-party applications will use multiple flows, it is not only necessary to account for sharing across applications, but also across flows belonging to the same application. In order to be able to consider the effects of "internal" sharing, Remos supports multi-flow queries in which the application lists all its flows simultaneously. Applications can generate flows with very diverse sharing characteristics, ranging from constrained low-bandwidth audio to bursty high-bandwidth data flows. Remos collapses this broad spectrum into three types of flows. *Fixed flows* have a specific bandwidth requirement. *Variable flows* have related requirements and demand the maximum available bandwidth that can be provided to all such flows in a given ratio. (e.g., all flows in a typical all-to-all communication operation have the same requirements) Finally, *independent flows* simply want maximum available bandwidth. These flow types also reflect priorities when sufficient resources are not available to satisfy all the flows. *Fixed flows* are considered first, followed by *variable flows*, then *independent flows*.

Determining how the throughput of a flow is affected by other messages in transit is very complicated and network specific. Remos approximates this complex behavior by assuming that, all else being equal, the bottleneck link bandwidth is shared equally by all flows (that are not bottlenecked elsewhere). If other information is available, Remos can use different sharing policies when estimating flow bandwidths. The basic sharing policy assumed by Remos corresponds to the max-min fair share policy [4]. use topology-based queries are themselves responsible for taking the effects of both internal and external sharing into account..

3.3 Accuracy

Applications ideally want information about the level of service they can expect to receive in the future, but most users today must use past performance as a predictor of the future. Different applications are also interested in activities on different timescales. A synchronous parallel application expects to transfer bursts of data in short periods of time, while a long running data intensive application may be interested in throughput over an extended period of time. For this reason, relevant queries in the Remos interface accept a timeframe parameter that allows the user to request data collected and averaged for a specific time window.

Network information such as available bandwidth changes continuously due to sharing and as a result, characterizing these metrics by a single number can be misleading. For example, knowing that bandwidth availability has been very stable represents a different scenario from it being an average of rapidly changing instantaneous bandwidths. To address these aspects, the Remos interface adds statistical variability and estimation accuracy parameters to all dynamic quantitative information. Since the actual distributions for the measured quantities are generally not known, we present the variability of network parameters using quartiles [5].

3.4 Implementation

An initial version of Remos API has been implemented. It has two components, a Collector and Modeler, that are responsible for network-oriented and application-oriented functionality, respectively. The Collector is responsible for collecting low-level network information. Such data can be collected in many ways, e.g. one can periodically run benchmarks that probe the network for available bandwidth or rely on information collected by applications [8]. Our current implementation uses a third method: the Collector explicitly queries routers using SNMP [2] for both topology and dynamic bandwidth information. This is a standard way of monitoring networks and it should allow us to collect detailed information in a relatively non-intrusive way on a broad set of networks. The Modeler is a library that is linked with the application. It translates the information provided by the Collector into a logical topology graph or per-flow data in response to application requests. The Modeler-Collector architecture is in part driven by the need to address scalability issues and network heterogeneity. In large networked environments, multiple Collectors may have to be deployed and each Collector can collect information in a way that is most appropriate for the network it is responsible for. Work is in progress on implementing Collectors that use sources of network information other than SNMP.

	Explicit	Implicit
Indirect (continuous)	Queres to network management database Rate management cells	Counting lost packets
Event-driven (direct)	Handlers to react to changes in the network	Reacting to lost retransmissions

Figure 1: Examples for two dimensions of application/network coupling.

Some initial experiments using the Remos interface on a dedicated IP-based testbed at Carnegie Mellon University are described in [6].

4 Concluding remarks

Figure 1 gives examples of the 4 different kinds of coupling between applications and networks that we discussed. Network-ware applications today focus overwhelmingly on implicit interaction. We argue that this state of affairs is due to the current (lack of) support for other interactions models by network architectures. As network architectures begin to provide information that is more accurate, more timely, and more detailed, network-aware applications will be motivated to also explore explicit interaction. E.g., an application may at startup use explicit queries to obtain information about available nodes and links, and then use implicit feedback after the startup phase.

Acknowledgments

We appreciate contributions and comments by the Remulac and Darwin groups.

Effort sponsored 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.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

References

[1] J. Bolliger and T. Gross. A framework-based approach to the development of network-aware applications. *IEEE Trans. Softw. Eng.*, 24(9):(to appear), Sept 1998.

[2] J. Case, K. McCloghrie, M. Rose, and S. Waldbusser. Protocol operations for version 2 of the simple network management protocol (snmpv2), January 1999. RFC 1905.

[3] T. DeWitt, T. Gross, B. Lowekamp, N. Miller, P. Steenkiste, J. Subhlok, and D. Sutherland. Remos: A resource monitoring system for network-aware applications. Technical Report CMU-CS-97-194, Carnegie Mellon School of Computer Science, Dec 1997.

[4] J. M. Jaffe. Bottleneck flow control. *IEEE Transactions on Communications*, 29(7):954–962, July 1981.

[5] R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, Inc., 1991.

[6] B. Lowekamp, N. Miller, D. Sutherland, T. Gross, P. Steenkiste, and J. Subhlok. A resource query interface for network-aware applications. In *Proc. 7th IEEE Symp. High-Performance Distr. Comp.*, page (to appear), July 1998.

[7] P. Steenkiste. Middleware adaptation models for network-aware application. Technical report, Department of Computer Science, Carnegie Mellon University, June 1998.

[8] M. Stemm, S. Seshan, and R. Katz. Spand: Shared passive network performance discovery. In *USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, June 1997.