Structuring QoS-Supporting Services with Smart Proxies

Rainer Koster and Thorsten Kramp

Distributed Systems Group, Dept. of Computer Science
University of Kaiserslautern, P.O. Box 3049, 67653 Kaiserslautern, Germany
{koster,kramp}@informatik.uni-kl.de
http://www.uni-kl.de/AG-Nehmer/Projekte/Squirrel

Abstract. While middleware platforms have been established in best-effort environments nowadays, support for QoS-sensitive services is still found lacking. More specifically, due to the high diversity of QoS requirements, the abstractions provided for QoS-unaware services cannot be maintained and the developer has to face the difficulties of low-level networking in heterogeneous environments again. In this paper, we therefore propose the notion of *smart proxies* as an effective means for making the use of QoS-sensitive services for the client-application developer as comfortable as the use of QoS-unaware services. This is achieved without imposing restrictions on the internal mechanisms and protocols used by an QoS-sensitive service to guarantee an agreed on level of QoS. Basically, smart proxies encapsulate service-specific code which is downloaded dynamically to the client during binding establishment. The benefits of this model are discussed in general and exemplified in a case study.

1 Introduction

Today's middleware platforms such as CORBA [17], DCOM [2], and DCE [4] have emerged as key components in heterogeneous environments with best-effort requirements. For QoS-sensitive application domains, however, the abstractions provided are still insufficient at best and prohibitively unsuitable at worst. In general, middleware platforms allow developing client and server applications independently of each other, with abstract interface specifications that are written in a language-independent *interface definition language* (IDL) and represent the link between client and server programmers. Aside from the interface specification neither the client programmer needs to know how the servers used by his client are implemented nor the server programmer needs to know about the internals of the clients that will access her server. Stubs, skeletons, and communications protocols in concert, directed by the middleware core, shield the client programmer from the low-level details of heterogeneous networking. Moreover, while performance issues, additional failure modes, and restricted parameter-passing rules tell the client that a service might be remotely located, it remains unaware of the exact location of the service.

This model works well for best-effort application domains, yet applications with more stringent QoS requirements are not adequately supported. In fact, considering the vast diversity of QoS requirements imposed on middleware platforms by, for example, next-generation multimedia applications or within mobile environments (Fig. 1), it is highly unlikely that a single middleware platform will meet all these requirements equally well — specifically since, besides traditional remote invocations, time-constraint messages and continuous media streams are becoming more important. The latter demand protocols with predictable latency and strictly controlled jitter while the respective mechanisms necessarily vary with the underlying network technologies. For example, with a QoS-supporting network such as ATM, bandwidth reservations can be easily mapped onto native network parameters, whereas a best-effort network requires a feedback mechanism and buffering on top to effectively control the stream according to given QoS constraints. Moreover, bandwidth limitations often enforce data compression with an appropriate codec whose choice also depends on the processing time and buffer space available at the client and the server. Consequently, CORBA, for instance, does not attempt to define a one-size-fits-all streaming protocol but only prescribes generic control and management interfaces for streams [16]. Since QoS inherently is an end-to-end issue, this leaves it to both the client *and* the server application-developers to implement the low-level protocols needed which, of course, cannot be generated automatically from an IDL description.

In this paper, we therefore propose the use of *smart proxies* as a structuring mechanism for QoS-sensitive services. Basically, any service-specific code needed at the client side is encapsulated in a smart proxy, which replaces the traditional stub and provides to the client the same high-level service-centred interface as if its remote server would be co-located with the client. This high-level interface, in turn, can be described in an IDL and may provide for high-level QoS negotiation in terms of, for example, frame rate or resolution, leaving it to the smart proxy and the server to map service-level parameters to corresponding resource requirements and low-level mechanisms. Access to a QoS-supporting service then becomes as easy for the client programmer as it is to a conventional service that does not need particular QoS provisions.

As a consequence, all low-level service-specific development efforts are shifted to the service developer, who implements both the server and its smart proxies using whatever protocol functionality and communications patterns are appropriate. Transparently to the client, different implementations may be tailored for particular environments. The service interface, however, is unaffected by the internal implementation and remains constant, shielding the client from all low-level technicalities. Of course, while the client so far can be implemented without knowing what service implementations it will connect to, all eligible proxies still must be available at the client side beforehand. We therefore propose that smart proxies are loaded dynamically from the server during binding establishment. The middleware platform downloads the smart proxy to the client machine and dynamically links its code to the client application on demand; from then on, the smart proxy handles the communication to its server.

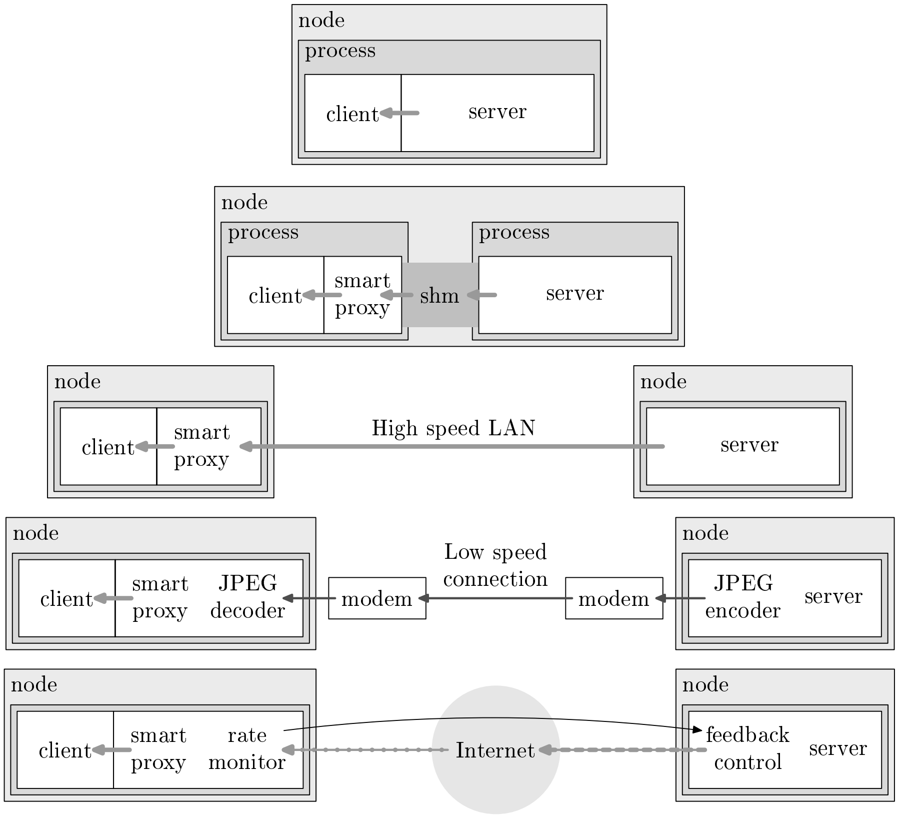

Fig. 1. Various Communication Scenarios

The remainder of this paper is structured as follows. Section 2 introduces the notion of smart proxies in more detail. Sections 3 and 4 then discuss the effects of downloading smart proxies dynamically at run time and what support is required from the middleware platform, respectively, followed, in Section 5, by a case study. Related work is summarised in Section 6 before the paper closes with conclusions and a brief outlook on future work in Section 7.

2 Smart Proxies

A *smart proxy* is service-specific code added to a client application. The client communicates with the smart proxy — and, thus, indirectly with the remote server — through a local interface as it would do with a co-located server, including QoS negotiation where appropriate. Whether the server is actually located within the client address space, on the client machine, or on a remote machine

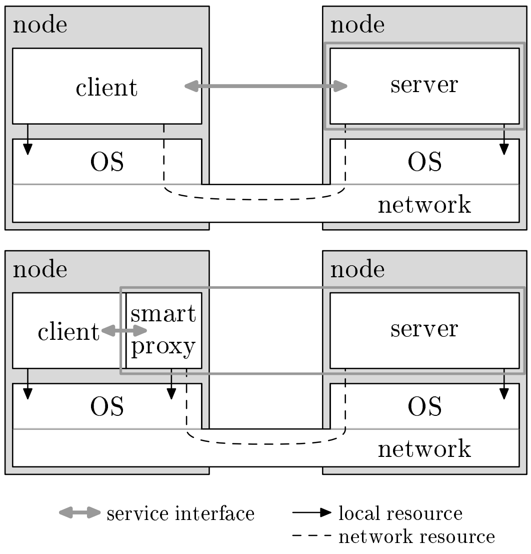

Fig. 2. Service Abstractions

is transparent to the client (performance issues, additional failure modes, and slightly restricted parameter passing rules aside).

For simple communication mechanisms, this functionality is identical to that provided by stubs automatically generated from an IDL description, as employed in CORBA or DCE, for instance. These stubs are limited to marshalling and unmarshalling of parameters, and sending requests via the standard protocols provided by their request broker. A smart proxy, in contrast, can also provide arbitrary functionality such as compression and sophisticated service-specific QoS management, although it must be developed specifically for each service instead of being automatically generated.

Since there is no one-size-fits-all protocol for applications that require particular performance optimisations or QoS functionality, it is highly unlikely that a single platform can satisfy all these requirements. In cases in which the platform's default protocol proves insufficient, client and server then need to communicate directly, that is, the low-level protocol used by the service actually becomes the service interface or at least part of it (Fig. 2). Hence, server implementations providing the same functionality but using different internal protocols essentially become different services and require different client software. This problem can be mitigated by supporting a set of protocols at both sides and choosing one in both sets during connection establishment. The CORBA telecoms specification relies on this approach for streaming by merely defining compatibility of stream endpoints [16]. The client, however, still needs to know how the internal communication works, what capabilities are provided by the server, and where the server is located.

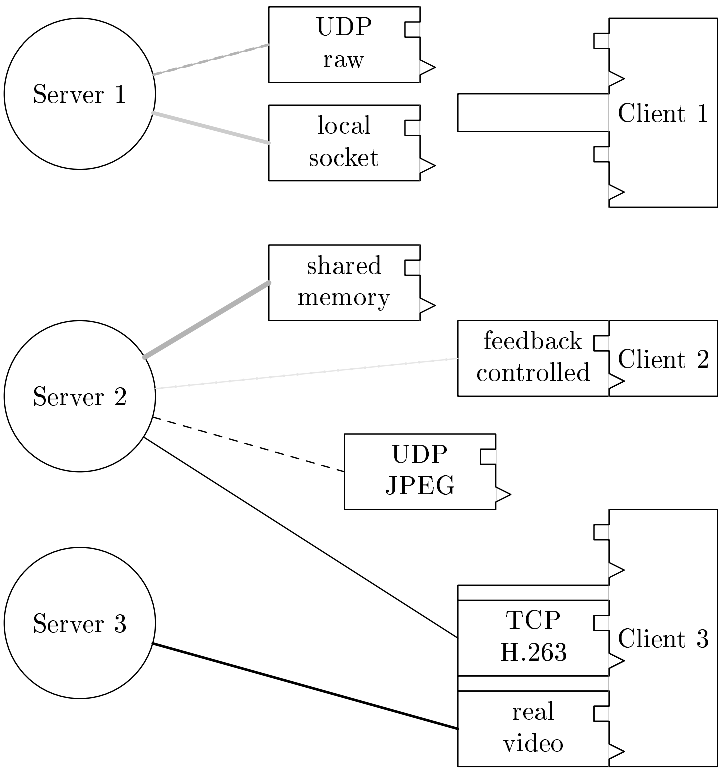

Fig. 3. Exemplified Smart Proxy Usage

By integrating and hiding complex communication mechanisms in smart proxies, the level of abstraction as provided by middleware platforms for QoS-unaware remote invocations can also be achieved for QoS-supporting services. There may be a variety of specialised server implementations and servers may provide different communication mechanisms for different connections (Fig. 3). For instance, a server could use shared memory locally, a compression mechanism and UDP across the Internet, or raw Ethernet on a dedicated LAN. In any case, the corresponding smart proxy implements the client side of the communication link, while, from the point of view of the client application, offering the same service-oriented high-level interface.

Note that the development of a smart proxy is not an additional effort. If complex functionality is required at the client and the server to appropriately handle a connection, this functionality unavoidably must be implemented manually and cannot be automatically generated from an abstract interface specification. Without smart proxies, however, the client developer as well as the service developer need to know the details of the communication protocol, whereas with smart proxies, the low-level details are hidden from the client developer and only the service developer, who is more likely to be familiar with low-level aspects of the service anyway, implements the low-level communication with the additional benefit of having both ends of the connection under control. The latter is particularly important for QoS control, which inherently is an end-to-end issue; in this case, both sides of a client/server connection must tightly cooperate to provide the negotiated level of QoS.

Consider, for instance, a video-streaming service. Its interface could include some high-level QoS parameters such as frame rate and resolution. With smart proxies the mapping of different QoS settings to low-level resource reservations and communication protocols can be handled transparently for the client developer:

1. If the underlying system supports resource reservation, the smart proxy can map these parameters to low-level resource requirements for the local node and the network connection in terms of CPU capacity and network bandwidth, for instance. Then, the smart proxy can obtain local resources and negotiate a guaranteed QoS with the server. This QoS then is reported to the client application, again in terms of high-level parameters such as frame rate and resolution.
2. If only a best-effort transport protocol is available, sophisticated feedback mechanisms are frequently used for QoS adaptation to guarantee an agreed on level of QoS. Such feedback loops, however, can be employed internally between the smart proxy and server without affecting the client application.
3. If the client and server happen to be in the same address space, no smart proxy is needed at all and the client can directly negotiate with the server what quality can be achieved with the resources available at this node. Substituting the server for the smart proxy again is transparent for the client since both share the same high-level interface.

In each of these example scenarios, the actual QoS management is hidden from the client application, whereas, without smart proxies, every client would need to handle all these cases itself. As a consequence, each client developer usually would have to implement the functionality for each connection type and server implementation himself.

Furthermore, smart proxies can also be beneficial for improving other non-functional aspects. They may, for instance, implement caching or prefetching strategies, which require service-specific knowledge about access patterns and appropriate consistency models.

Finally, service updates that affect communication protocols usually require updates of the client software as well, even if the high-level service interface is unaffected by the update. While a new service additionally can implement the old protocol, the benefits of the new features cannot be utilised in backward compatibility mode. Smart proxies, in contrast, are developed along with the server, and simply need to be replaced when updating a service without affecting the actual client application.

3 Proxy Shipping

In general, smart proxies and clients can be shipped either *statically* or *dynamically*. With the static approach, the smart proxy is somehow sent to the client developer out of band (e. g., via email) and linked with the application. Hence, all smart proxies that a client might need must be present before the service is accessed. NETSCAPE plug-ins [3], for instance, work in a similar way. While this is a more systematic approach than integrating smart-proxy functionality directly into the client software, it only partially realises the benefits of smart proxies.

Dynamic proxy shipping, in contrast, is much more flexible. The most appropriate server, and then the most appropriate smart proxy of this server can be chosen and shipped to the client during binding establishment, taking the current resource availability into account. Moreover, as long as the service interface remains stable, server updates become completely transparent to the client: the updated smart proxy is simply sent and communicates with the new server version. At the client side, only smart proxies currently in use need to be present at the client.

However, regardless how a smart proxy is shipped, in an heterogeneous environment it must be available in several versions. If a server has m different types of smart proxies supporting different types of network connections, and n types of client applications run on p platforms, the service developer effectively must implement $m \times p$ smart proxies, where p is typically small. Note that there is still a lot less effort than implementing the same functionality in $n \times m \times p$ client versions. Furthermore, since smart proxies are implemented by the service developer, often only a recompile or minor modifications are needed for different platforms. If the development is spread among the client developers, in contrast, the same functionality generally would have to be re-invented and implemented over and over again.

To implement dynamic loading of smart proxies, code must be shipped from the server to the client. One way to allow this is using a virtual machine (such as provided by Java) for running the smart proxies. In this case, proxies need to be programmed only for the virtual machine, but not for each possible client platform in an heterogeneous environment. Yet, since smart proxies are meant to perform computationally intensive tasks such as decoding video frames and dealing with real-time constraints, run-time efficiency and predictability are important issues. Although considerable effort is being spent on improving virtual machines to this respect, the current state of the art is hardly satisfactory. Hence, we have explored a different approach. Many systems allow dynamic linking of shared libraries at run time. Then, smart proxies can be built as native-code shared libraries that are sent over the network and are dynamically linked to the client. While this mechanism requires a different smart proxy version for each supported client platform, language heterogenity is achieved to a certain extent since many languages share the same object-code format (e. g., ELF shared libraries [19]) and, thus, can be linked to libraries written in another language.

Code shipping in general, however, also causes serious security risks which are far easier to control with a virtual machine that ‘sandboxes’ smart proxies and, thus, protects the client process from malicious operations. With native-code libraries the problem is much more difficult and requires future research. For now, the problems may be mitigated by using only trusted servers and signing smart proxies cryptographically, or restricting the use of smart proxies to security domains such as a cluster of computers.

4 Platform Support

Smart proxies as a structuring mechanism can be used without any particular support from an underlying middleware platform. Developing services as a combination of a server and smart proxies first of all is a reasonable way of building distributed services. The smart proxies define the interface to the service from a client developer’s point of view and only need to be linked to clients that want to use it.

For dynamic proxy shipping, in contrast, client and server at least must be able to transmit the proxy at connection setup and link it to the client application. With Java, the virtual machine handles downloading the byte code of smart proxies and running it, whereas the use of shared libraries with a language such as C or C++ is slightly more difficult. Since the function definitions are not available at compile time of the client, functions of smart proxies must be called via function pointers. The machinery required for these indirections, however, can be generated automatically from the header file of a smart proxy for which we have developed a tool. Apart from this, there needs to be a standardized way of retrieving the smart proxy from the server. The client may open, for example, a TCP/IP connection to the server and download the smart-proxy code to a local disk prior to linking it to the application code using the operating system’s default dynamic-linking facilities such as `dlopen` under Unix. We have implemented this mechanism on LINUX and extended `dlopen` to read the code to be linked directly from the network rather than from a file.

However, the functionality required for dynamically downloading and linking smart proxies should be integrated with a middleware platform to be readily available. The middleware platform then is responsible for the handling of service references and locating the respective servers as well as performing the the actual shipping and linking of smart proxies. For choosing the best-suited smart proxy for a given client, the middleware platform automatically would report the client’s operating system, hardware platform, and network technology to server, possibly complemented by status information such as the current processing load. It may also be useful to establish smart-proxy repositories to keep smart proxy and server implementations consistent. Finally, a middleware platform could provide some security in loading proxies such as checking their integrity.

To this end, we are currently investigating how smart proxy support can be integrated with CORBA. The ability to access objects by value [15] provides some prerequisites for proxy shipping and allows the development of a CORBA service for this task. For continuous-media transmission, smart proxies can be built along the lines of the CORBA telecoms stream-management specification [16], which provides services with the ability to exploit protocols not directly supported by the ORB itself. To fully utilise the potential of smart proxies with respect to QoS, however, the underlying operating environment including the operating system, the networking subsystem, and the middleware platform must support some resource management. The system should at least provide mechanisms for smart proxies and servers to reserve elementary resources such as CPU cycles and memory buffers on the respective node, and to specify connection properties such as guaranteed bandwidth and maximum latency. In this context, we are currently developing an open low-level foundation for QoS-supporting middleware in combination with appropriate operating-system-level support [6,7,8,9].

5 Case Study

To demonstrate the benefits of using smart proxies, we have implemented a live-video service providing access to a camera and to be used by, for example, video-conferencing and video-surveillance clients. With this example application we can demonstrate the following important features of smart proxies:

- ▷ Different service implementations that use different communication mechanisms can be accessed transparently by clients through a uniform interface.
- ▷ Different QoS management strategies can be encapsulated in proxies.
- ▷ Different client applications using the same service all can utilise the set of protocols supported by the service's smart proxies without re-implementing endpoint functionality in each client.

Right now, we have not implemented the example on a middleware platform, but have used the modified `dlopen` mentioned above to prove the general feasibility of smart proxies.

The video server runs on x86 PCs with Linux 2.2 using a Hauppauge frame-grabber card with a camera as the video source. Clients and smart proxies have been implemented in C++, according to the following service interface:

```
class live_video {
   public:
      void start(int frame_rate);
      void stop();
      void get_frame(char* &frame, struct timeval &when);
      void free_frame(char* frame);
      int request(int frame_rate);
};
```

Calling the `start` method initiates the transmission of video frames with a given frame rate, calling `stop` terminates the transmission. Frame data can be read by calling `get_frame` which blocks until a frame becomes available and also reports the recording time of each frame returned. Finally, frames need to be freed with `free_frame`. The only QoS parameter controlled by this simple example is the frame rate. A client can try to make a reservation for some level of QoS using the `request` method. The frame rate returned can then be guaranteed by the system with a return value of 0 indicating that only best-effort access is supported.

5.1 Various Communication Mechanisms

We have implemented smart proxies and servers for four ways of communication. A simple UDP transmission just sends the frames over the network. Since UDP is unreliable and does not preserve order, packet losses and out-of-order delivery must be handled correctly. A second proxy-server pair also uses UDP but employs JPEG compression to save network bandwidth at the expense of higher computational load. The third version uses the B_EA_T protocol for local networks [6], which is reliable and provides some level of QoS guarantees discussed in more detail below. Finally, proxy and server can efficiently communicate via shared memory if client and server happen to be co-located on the same node. Based on information submitted by the client, the server chooses the smart proxy which most closely matches the client requirements and is compatible with the processing time and network bandwidth available.

The various performance characteristics of the different communication protocols are illustrated in Fig. 4. Transmitting video frames over an idle 10 Mbps Ethernet peaks at 8.5 fps (frames per second) for raw images and 12.8 fps for JPEG-encoded images, independently of the network protocol used. For a co-located client/server pair, finally, shared memory reaches the expected frame rate of 25 fps.

5.2 QoS Management

The B_EA_T protocol provides deterministic network access on an Ethernet and a means for bandwidth reservation, which is used as a simple example for QoS management. When a client application requests a particular frame rate from its service, a best-effort smart proxy (i. e., one that relies on UDP) would simply return 0 to indicate that there are no guarantees available. With B_EA_T, in contrast, a smart proxy could map the high-level parameter ‘frame rate’ to the low-level parameter ‘bandwidth’. This mapping can be done as part of the service logic since the smart proxy knows the size of the frames used. Then, the smart proxy tries to reserve this bandwidth with the transport protocol and checks with the server what frame rate can be delivered from the video source. Finally, the frame rate that can be guaranteed by server and network is reported to the client, which is unaware of the required low-level resource reservation and mechanisms used to guarantee the frame rate.

The advantage of using a resource-reservation protocol shows when transmitting video frames in competition with a synthetically generated load of 4 Mbps (Fig 5). Plain UDP peaks at a frame rate of only 4.3 fps for raw images, whereas B_EA_T still allows up to 6 fps for raw images and about 12 fps for JPEG encoded

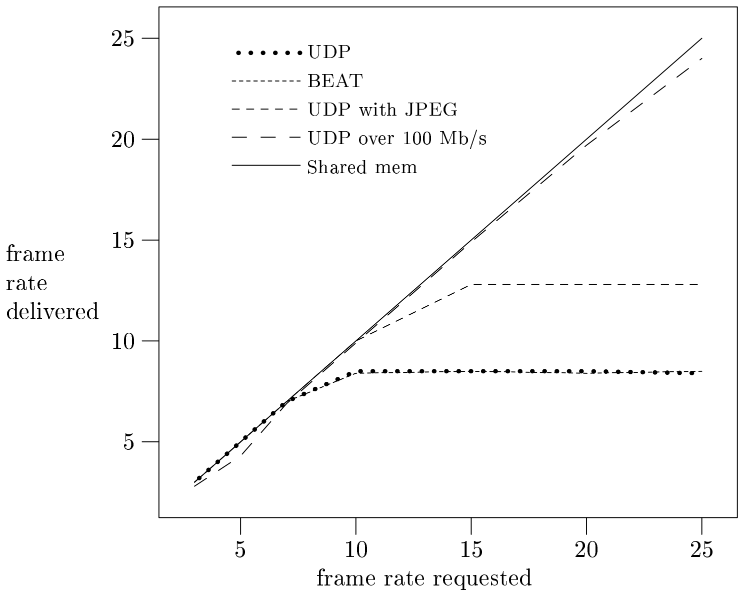

Fig. 4. Different Communication Protocols

ones. While JPEG encoding induces an additional computational load it allows to submit a reasonable frame rate even if the available network bandwidth would not allow an uncompressed transmission.

In a similar manner, reservations with RSVP or other protocols could be encapsulated. For this case study, we have also assumed that network bandwidth is the only potentially scarce resource. More elaborate smart proxy/server combinations would also control other resources on the client and the server side such as processing time or buffer space, as well as additional QoS parameters such as jitter and latency bounds.

Even if there is no support for reservations, smart proxies can improve QoS. Advanced best-effort transport protocols for continuous media typically employ some feedback mechanism to adjust the send rate of the server to the resources actually available [1,18]. The client-side code required for these features again can be provided by smart proxies without modifying the client.

5.3 Proxy Reuse for Several Clients

As one example for the versatility of our smart proxies, we have used the live-video service twice in our teleconferencing clients. A local server shows the picture of the person running the client while a remote server shows the person he or she is talking to. The client accesses both servers in the same way, relying on the smart proxies to take care of finding the best way of transmitting the video frames.

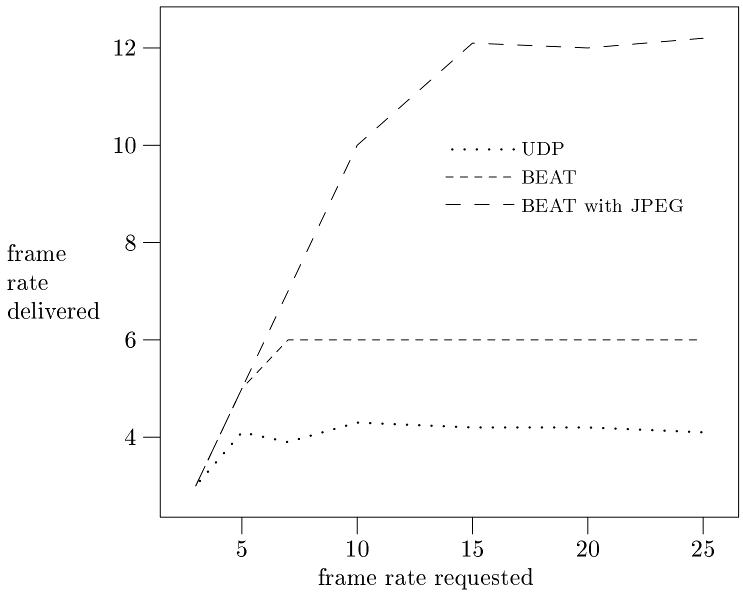

Fig. 5. Reservation with Smart Proxies

Furthermore, we have implemented a surveillance tool re-using the live-video server and its smart proxies. This tool connects to a remote video service and compares adjacent frames, raising an alarm when the picture changes. Regardless of what type of connection is best, the same service as for the teleconferencing can be used. Without smart proxies, all client-side functionality for the respective connection types would have been to be re-implemented. Even for this rather rudimentary example this would have resulted in a considerable effort in developing each client.

6 Related Work

The notion of smart proxies is most closely related to the work on *fragmented* or *distributed objects* as proposed by Makpangou et al. [12] and, more recently, within the Globe [20,21] and AspectIX [5] projects. The fundamental idea is to allow objects to be physically distributed and to consist of fragments at several nodes; distribution and communication between fragments are hidden from other (client) objects.

Globe is a middleware platform that employs distributed objects to provide scalability to wide-area distributed applications such as replication and caching for web documents. Middleware services are used to locate and download fragments, through which the object is accessed. The AspectIX project, in contrast, while also being based on distributed objects, is focussed on enhanced

QoS support, extending CORBA by support for mobile and reconfigurable object fragments.

Smart proxies can be seen as a particular way of using distributed objects and represent a simpler and more elementary model. In contrast to the symmetric model of fragments in a distributed object, however, smart proxies and servers have distinct roles. This approach is more similar to the familiar client and server model and, hence, may be more easily adopted by programmers than the development of servers as distributed objects. In addition, less platform support is needed. For instance, proxies do not have persistent state, are selected by the server, and need not be located independently of the server. Moreover, while we try to provide QoS support for applications such as continuous media streaming services, Globe focusses on scalability. It is not obvious, for instance, whether complex transmission mechanisms such as compression or feedback can easily be integrated with Globe's object fragments.

The QuO architecture [11,22] takes a different approach to hiding QoS management issues from the client application. Separately from the IDL defining the functional interface of an object, QoS parameters and adaptive behaviour are specified by QuO description languages. From these QDL so called delegates are generated and linked to the client application in a similar way as stubs are generated from the IDL. Hence, the delegates are basically a kind of statically shipped smart proxies. Compared to our approach, on the one hand, the QuO architecture and the automatic code generation facilitate integration of functionality such as resource reservation, QoS monitoring, and adaptation. On the other hand, complex delegate functionality not provided by the platform can only be added to the QDL as source code in the implementation languages of the clients, when the service interface is designed.

Within a more narrow context, the concept of embedding service-specific code within client applications has also been explored by Yoshikawa et al. with so-called *smart clients* [24]. These smart clients were primarily used to implement caching and prefetching as a means for increasing the scalability of Internet services in terms of performance, load balancing, and fault tolerance. Of course, these tasks can also be encapsulated in smart proxies.

Proxies are also an important concept in Sun's Jini environment [13,23], in which services are defined in terms of Java interfaces. To access a service, a lookup service returns sort of a smart proxy to the client that communicates with the server. Jini mainly uses this mechanism to allow devices and services to be dynamically added to and removed from the system. Of course, it also allows server and proxy to choose their own protocol for communicating with each other. Since Jini is based on Java, it inherits the advantages of security, ease of code shipping, and platform independence, as well as the drawbacks of being restricted to one language and the potential performance penalties and unpredictability of a virtual machine.

Furthermore, there are several ongoing efforts to develop QoS supporting middleware platforms in general and to improve real-time and QoS properties of CORBA in particular. Specifically related to QoS for continuous-media transmission are implementations of the CORBA telecoms specification [16] such as the audio/video streaming service built on top of TAO [14]. Work on configurable middleware platforms, finally, is related to our work since these platforms open a wider range of infrastructural support to smart proxies. TAO's pluggable protocol framework [10] is only one example of ongoing efforts in this context.

7 Conclusions

In this paper we have introduced the notion of smart proxies as an effective means for structuring QoS-sensitive services. The benefits of this approach are threefold. Firstly, service-specific client code is separated from the client application-code and encapsulated in self-contained modules. This leads to a clear separation of functionality as a prerequisite for dynamically substituting modules that adhere to the same high-level interface. Secondly, all low-level service-specific development efforts are shifted to the service developer, while the client developer merely interacts with a high-level interface in the same way as he does with QoS-unaware services. As a consequence, instead of developing low-level client-side functionality over and over again for each client application anew, with smart proxies this functionality is only developed once by the service developer who knows the internals of his service best anyway. Thirdly, dynamic shipping of smart proxies allows for the seamless introduction of improved service functionality or completely new service implementations without requiring modifications of the client applications. Only the functionality actually used must be available at the client.

To demonstrate the viability of smart proxies, we have implemented a video service that supports a small range of different communication protocols, namely unreliable UDP with and without compression, B_EA_T with explicit resource reservation, and shared memory for co-located client and servers. The video service is used both in a video-conferencing tool and a video-surveillance tool which only interact with the high-level interface of the video service. The use of smart proxies for low-level networking significantly reduced the development of both services and both services automatically would benefit from adding another smart proxy implementing, for example, a feedback loop over UDP.

As part of our future work we will integrate smart proxies with CORBA, making use of and possibly expanding on the recent *objects-by-value* specification. Furthermore, support for smart proxies will be integrated with our own QoS-supporting middleware under development [6,7,8,9].

Acknowledgements

We are indebted to Marcus Demker for implementing parts of the case study. Moreover, we thank the anonymous reviewers for their helpful comments.

References

1. S. Cen, C. Pu, R. Staehli, C. Cowan, and J. Walpole. A distributed real-time mpeg video audio player. In *Proceedings of the Fifth International Workshop on Network and Operating Systems Support for Digital Audio and Video*, volume 1018 of *Lecture Notes in Computer Science*, pages 142–153. Springer Verlag, April 1995. 283
2. Microsoft Corp. *Distributed Component Object Model Protocol*, 1998. 273
3. Netscape Communications Corporation. Plug-in guide. `http://developer.netscape.com/docs/manuals/communicator/plugin/index.htm`, January 1998. 279
4. The Open Group. *Introduction to OSF DCE 1.2.2*, November 1997. 273
5. F. Hauck, U. Becker, M. Geier, E. Meier, U. Rastofer, and M. Steckermeier. AspectIX, an aspect-oriented and CORBA-compliant ORB architecture. Technical Report TR-I4-98-08, Friedrich-Alexander-University, Erlangen-Nürnberg, September 1988. 284
6. R. Koster. Design of a real-time communication service for local-area networks. Diplom thesis, Department of Computer Science, University of Kaiserslautern, May 1998. 281, 282, 286
7. T. Kramp and G. Coulson. The design of a flexible communications framework for next-generation middleware. Technical Report SFB 501 12/99 and MPG-99-25, Dept. of Computer Science, University of Kaiserslautern, and Dept. of Computing, Lancaster University, 1999. 281, 286
8. T. Kramp and R. Koster. A service-centred approach to QoS-supporting middleware. Work-in-Progress Paper presented at *Middleware '98 (IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing)*, September 1998. 281, 286
9. T. Kramp and R. Koster. Flexible event-based threading for QoS-supporting middleware. In *Proceedings of the Second International Working Conference on Distributed Applications and Interoperable Systems (DAIS)*. IFIP, July 1999. 281, 286
10. F. Kuhns, C. O'Ryan, D. C. Schmidt, O. Othman, and J. Parsons. The design and performance of a pluggable protocols framework for object request broker middleware. In *Proceedings of the sixth IFIP International Workshop on Protocols for High-Speed Networks (PfHSN)*, August 1999. 286
11. J. P. Loyall, D. E. Bakken, R. E. Schantz, J. A. Zinky, D. A. Karr, R. Vanegas, and K. R. Anderson. QoS aspect languages and their runtime integration. In *Proceedings of the Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR98)*, volume 1511 of *Lecture Notes in Computer Science*. Springer Verlag, May 1998. 285
12. M. Makpangou, Y. Gourhant, J.-P. Le Narzul, and M. Shapiro. Fragmented objects for distributed abstractions. In T. L. Casavant and M. Singhal, editors, *Readings in Distributed Computing Systems*, pages 170–186. IEEE Computer Society Press, July 1994. 284
13. Sun Microsystems. Jini architectural overview, 1999. Technical White Paper. 285
14. S. Mungee, N. Surendran, and D. C. Schmidt. The design and performance of a CORBA audio/video streaming service. In *HICSS-32 International Conference on System Sciences, minitrack on Multimedia DBMS and WWW*, January 1999. 286
15. OMG. CORBA objects by value. `http://www.omg.org/cgi-bin/doc?orbos/98-01-18`, 1998. orbos/98-01-18. 280

16. OMG. CORBA telecoms specification. `http://www.omg.org/corba/ctfull.html`, June 1998. formal/98-07-12. 274, 276, 281, 286
17. OMG. *The Common Object Request Broker: Architecture and Specification (Release 2.2)*, February 1998. 273
18. L. A. Rowe and B. C. Smith. A continuous media player. In *Proceedings of the third International Workshop on Network and Operating Systems Support for Digital Audio and Video*, volume 712 of *Lecture Notes in Computer Science*, pages 376–386. Springer Verlag, November 1992. 283
19. SunSoft. SunOS 5.3 Linker and Libraries Manual, 1993. 279
20. M. van Steen, P. Homburg, and A. S. Tanenbaum. Globe: A wide-area distributed system. *IEEE Concurrency*, pages 70–78, January-March 1999. 284
21. M. van Steen, A. S. Tanenbaum, I. Kuz, and H. J. Sips. A scalable middleware solution for advanced wide-area web services. In *Proceedings of Middleware '98 (IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing)*, pages 37–53. Springer Verlag, September 1998. 284
22. R. Vanegas, J. A. Zinky, J. P. Loyall, D. A. Karr, R. E. Schantz, and D. E. Bakken. QuO's runtime support for quality of service in distributed objects. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98).* Springer Verlag, September 1998. 285
23. J. Waldo. The Jini architecture for network-centered computing. *Communications of the ACM*, 42(7):76–82, July 1999. 285
24. C. Yoshikawa, B. Chun, P. Eastham, A. Vahdat, T. Anderson, and D. Culler. Using smart clients to build scalable services. In *Proceedings of the USENIX 1997 Annual Technical Conference*, January 1997. 285