# Communication support for multiple QoS requirements

Hugo Miranda and Luís Rodrigues
Universidade de Lisboa
Edifício C5 - Campo Grande - 1700 LISBOA - Portugal

## Abstract

Many existing applications demand the simultaneous use of several communication channels with different Quality of Service (QoS) requirements. For optimal performance, the application designer should be able to to specify a protocol stack that meets hers/his QoS requirements through the composition of micro-protocols.

The paper gives examples of applications where the stacks supporting different QoSs are not independent. We show that these dependencies are hard to express with current communication architectures and propose a novel approach that supports a style of micro-protocol composition that satisfies both inter-QoS and intra-QoS constraints.

## 1 Introduction

Distributed applications are becoming increasingly complex, offering rich and powerful services to their users. In order to offer satisfactory performance, these applications are also becoming increasingly demanding in terms of communication support. It is easy to find applications that require the simultaneous use of several communication channels with different Quality of Service (QoS) requirements. Typical examples can be found in multimedia applications, where different channels for data, audio and video are used. The same sort of requirements can also be found in less obvious contexts. For instance, in replication management one might use different ordering properties for write and read operations.

The implementation of each QoS class requires the use of a specific protocol stack. For instance, network omissions are treated in a different way according to the type of information being transmitted. Depending on the compression algorithm, a video frame may be dropped, but a data lock request needs to be delivered in a reliable way. In order to avoid re-writing communication stacks for each new application, the new generation of communication services allow the application designer to specify a protocol stack that meets her/his QoS requirements through the composition of micro-protocols. We call the requirements that constraint the composition of the stack satisfying a single QoS an *intra-QoS* requirement.

In this paper we show that intra-QoS protocol composition is not enough and that existing communication architectures still lack some features that are essential to support applications with multiple QoS requirements. We provide examples of applications where the communication stacks that support different QoSs are not independent. Instead, the application requires the implementation of *inter-QoS* constraints that can be expressed in an elegant way if the communication stacks share common layers. We show that these dependencies are hard to express with current architectures and propose a novel approach that supports a style of micro-protocol composition that satisfies both inter-QoS and intra-QoS constraints.

The paper is organized as follows. In section 2 we motivate the problem by presenting some examples of the class of problems that impose inter-QoS constraints. Section 3 proposes a solution to the problem. Section 4 relates this work with several communication frameworks. Section 5 concludes this paper.

## 2 Motivation

In this section we provide examples of classes of problems that require the use of multiple QoS with

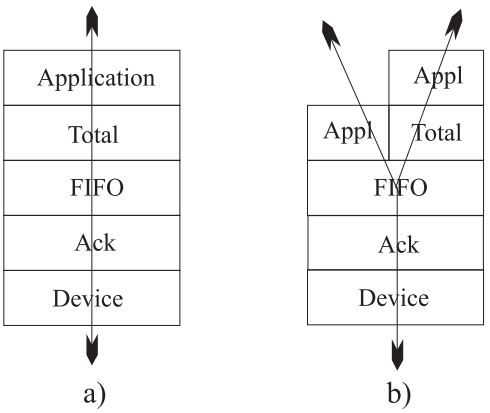


Figure 1: Two stack definition approaches: a) regular b) *cactus* like

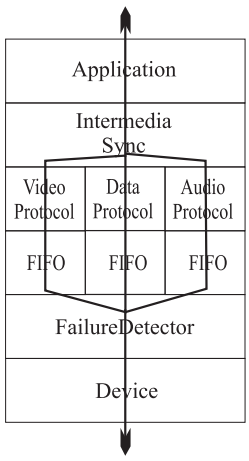


Figure 2: A multimedia stack with *diamond* shape

both intra-QoS and inter-QoS constraints. To illustrate a possible implementation of these requirements we will use stacks built from the composition of micro-protocols. Most recent architectures [10, 13, 5] support this type of composition, since modularization is essential to the development of powerful communication systems. Using protocol composition to obtain richer semantics simplifies the implementation and validation of protocols and allows applications to *pay* the services that they need instead of what the tool has to offer [6].

According to this approach, a stack can be built by "piling" layers of micro-protocols. Most existing systems support just independent stacks where messages have only *a single* path, i.e., messages cross all stack layers, getting all their properties. Some systems (e.g. Horus [14]) allow stacks to assume a *cactus* style, as illustrated in Figure 1, allowing messages to follow different paths when they are sent from different stack access points. The cactus can also be upside-down [13]. We will show that this type of composition is not rich enough to support complex QoS requirements.

## 2.1 Synchronized streams with different QoS

This example is representative of multimedia applications. To support interaction, the application opens different communication channels, one for each type of media (namely audio, video and data). These two streams require different transport protocols, thus different communication stacks are used. However, all streams need to be synchronized. To achieve this goal, an inter-stream synchronization layer can be used as proposed in [2].

Another problem of using different stacks is that failures can be detected in a non-consistent way by the protocols in each stack. The Maestro [1] system tackles this issue by creating a "core" stack in charge of coordinating the others. The core stack coordinates by sharing joins, leaves and failure detectors.

An elegant solution based on protocol composition could use the stack of Figure 2. This approach consists of having a common failure detection layer at the bottom of each stream and a synchronization layer in the upper layers. The combination of the several "paths" creates a "diamond" structure. This type of structure can be found in other systems [11] but as we will show, with a limited scope.

## 2.2 Operations with different semantics

When making remote invocations on distributed objects, the performance is improved if the appropriate QoS is used. Often, different operations have different semantics. We have already referred the case of read and write operations. Usually writes need to be totally ordered while reads are commu-

tative. Several other operations share this type of constraints. We will use the following example presented by Georges Brun-Cottan in the Ensemble [5] mailing list. Consider one application that needs to implement a distributed semaphore. A distributed fault-tolerant semaphore can be implemented as a replicated shared variable satisfying the following constraints:

- *Test & set* operations are delivered in FIFO and Total order with regard to each other to every group member.
- Release operations may be delivered only in FIFO order with regard to each other and with regard to *test & set* ones.

One way to satisfy these constraints is to build a new layer that offers two different QoS such as the one presented in [12]. Another solution is to achieve the same semantics through protocol composition. Having a single stack (similar to the stack of Figure 1a) imposes unnecessary delays, since release operations are also totally ordered.

The cactus like stack (Figure 1b) does not satisfy the constraints also: in this case the process would handle two stack access points, a structure that cannot enforce preservation of the desired FIFO order. The problem is the following: suppose that some process $p$ sends a *test & set* message $m_1$ followed by a release message $m_2$. Since $m_1$ is a totally ordered message it might be blocked within the total layer, waiting for previous message from some other process. This would allow message $m_2$, which is not forced to go through the total layer, to be delivered to the application before $m_1$.

A correct composition would require a structure such as the one depicted in Figure 3. The reader will notice that the protocol "path" associated with each QoS forms a "diamond" like structure similar to our previous example.

### 2.3 Discussion

The previous examples share some common properties that can be identified:

1. Several QoS channels need to be supported;
2. these channels are not independent of each other;

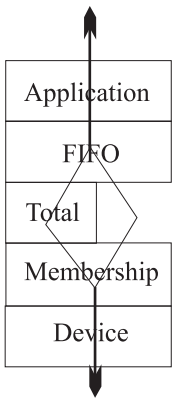


Figure 3: A stack for a distributed semaphore with *diamond* shape

3. the resulting protocol composition exhibits a "diamond" like structure.

These characteristics are not restricted to the examples above and can be found in other contexts. In the following section we will discuss mechanisms to support the development of protocol structures with these properties.

## 3 A protocol fabric

Protocol composition must be supported by a number of abstractions, tools and run-time mechanisms. Relevant run-time mechanisms include memory management for message structures, timeout management, thread management, etc. A classical example of work in this area is the $x$-kernel [8], which provides a powerful set of features to develop communication stacks. From the point of view of protocol composition, the function of the *protocol kernel* is to support the exchange of events between adjacent layers. This paper focus exclusively on the layer composition aspects of protocol kernels and does not addresses orthogonal issues such as buffer management, timer management, etc.

It should be noted that, in order to maintain layer independence, the micro-protocols should not be aware of the way they are interconnected. No explicit reference to a particular upper or lower layer should be allowed. Instead, each protocol should only invoke generic "up-event" and "down-event" calls. The events are delivered to the appropriate

protocols by the kernel according to the bindings established when the stacks are created (possibly, at run-time). We propose a novel architecture that allows the routing of events to be based not only on QoS parameters (intra-QoS constraints) but also on inter-QoS constraints. The protocol kernel acts as a switching fabric, routing messages between layers and ensuring that a path satisfying the desired QoS is followed.

## 3.1 The model

We define a *layer* as the implementation of a protocol. All protocols implement the same *event interface*, which defines the types of events each one is able to consume and produce. The format and semantics of these events is irrelevant to our exposition. Typical examples of events are data transmission requests, indication and confirmations. Examples of layers are "datagram transport", "positive acknowledgment", "total order", "checksum", etc. Good examples of relevant layers and events in the context of fault-tolerant applications can be found in [5].

We define a *session* as an instance of a layer [9]. The session maintains state that is used by the layer code to process events. A layer that implements ordering may keep a sequence number or a vector clock as part of the session state. In connection oriented protocols, the session also maintains information about the endpoints of the connection. Note that it is often useful to maintain several active sessions for the same layer even when they share the same endpoints: for instance, one might want to have different FIFO channels for different priority streams.

The clear separation between layers and sessions is key to our design. These two concepts can be combined to satisfy multiple QoS requirements as follows:

A *QoS* is defined as an ordered set of layers. The QoS defines which protocols must act on the messages and by which order they must be traversed. A *channel* is an instantiation of a QoS and is[1] characterized by a stack of sessions of the corresponding layers. For sake of clarity we assume that all QoS that interact with the application are terminated

[1] At a finer level of granularity, one could also consider different parameterizations of each protocol as an independent QoS.

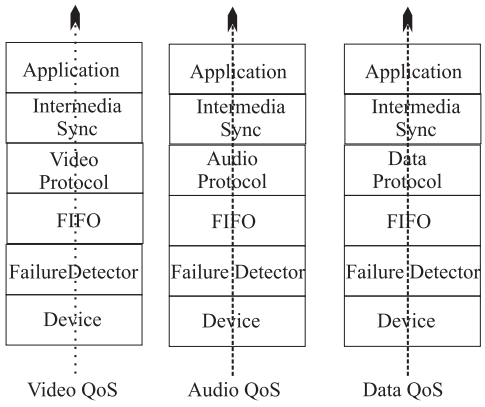


Figure 4: Inter-stack QoS requirements in QoS set

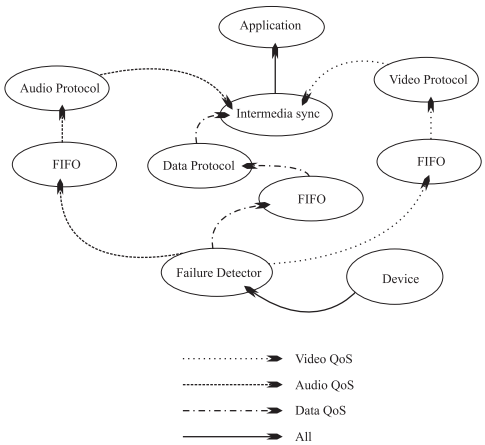


Figure 5: Inter-stack QoS requirements in acyclic graph

on top by a layer that we simply call the "Application". A session of the Application layer is also called an *endpoint*. A *stack* is a set of non-disjoint channels (i.e., each channel shares at least one session with some other channel of the same stack). Figures 4, 5, 6 illustrates these concepts.

## 3.2 Requirements revisited

We can now revisit the requirements of applications demanding the management of multi non-independent quality of services in face of our definitions. The requirements are presented in Table 1.

Requirement RQ-1 is the basic motivation for our

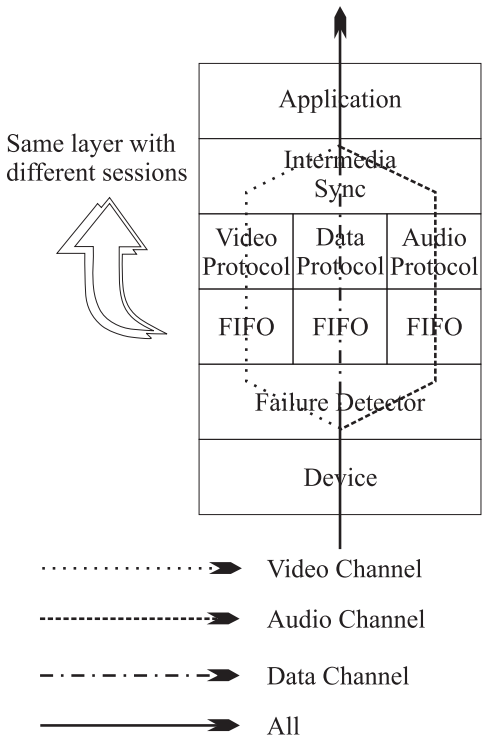


Figure 6: Inter-stack QoS requirements in stack with three channels

service. It is satisfied by allowing users to send messages over different channels. Requirement RQ-2 reflects the fact that, at the recipient site, the message should be processed by the peer entities of the protocols transversed during transmission. Requirement RQ-3 imposes that inter-QoS constraints need to be taken into account and stacks with "diamond" structures need to be built. Finally, in requirement RQ-4, we state that micro-protocols should be combined without the need to re-write any code. This preserves the flexibility of the system, allowing an existent platform to be *extended* instead of *recreated*.

### 3.3 The protocol kernel

The protocol kernel is responsible for maintaining meta-structures about the active stacks and for supporting exchange of events between adjacent layers. When an event is pushed down or popped up in the stack, it has to be "routed" to the appropriate session of the appropriate layer. This routing is sometimes performed in a different way for descending and ascending events.

RQ-1 The user should be allowed to send a message using different Qualities of Service.

RQ-2 The message should follow peer channels in different nodes.

RQ-3 Channels may or may not share sessions of common layers. In particular, they may share the same endpoint.

RQ-4 Micro-protocols independence should be preserved.

Table 1: Implementation requisites

To illustrate how associations between adjacent layers can be managed, consider the following "traditional" implementation of a stack containing PRESENTATION, connection oriented TRANSPORT and NETWORK layers (our example is generic, and independent of any specific protocol family). When a message is sent (descending event), the presentation layer selects a given session of the transport protocol. It then calls a function of the transport layer specifying the session in a parameter of the call. The transport adds a header to the data message containing a connection identifier and pushes the data further down by calling a function of the network layer. When the message is received (ascending event), the network layer reads the network protocol header, which indicates that the event must be routed to the transport layer. The network layer is unable to read the transport header so he cannot forward the message directly to the "session". Instead, a generic function of the transport protocol is called to extract the connection identifier from the transport header. The connection identifier is then used to obtain the correct transport "session" state, which is needed to process the event.

The previous approach has several drawbacks. To start with, it implements a form of static binding, where each layer is aware of its adjacent layers. For instance, if a new TRANSPORT2 is added to

the system, the code of the network layer needs to be modified in order to exchange events with that layer. This strongly limits the modularity of the system. Additionally, it requires "session" identifiers to be added and extracted from headers at all layers where this information is needed.

In our case we propose to use a scheme inspired in the Horus [13] and Ensemble [5] systems, where events between adjacent layers are routed based on "tags" associated with the events. Tags are obtained from QoS and inter-QoS constraints. In the Ensemble system these tags are associated with stacks, since Ensemble only supports a one-to-one mapping between stacks and sessions. In our case the tags are associated with channels and the kernel meta-structures register which sessions are transversed by a given channel. The technique has similarities with tag-switching techniques used in network routing (for instance in ATM switching) and offers similar advantages: events can be delivered immediately to the appropriate session and both space and processing time is saved by avoiding redundant headers.

### 3.4 Stack interface

In this paper we assume that every stack interfaces the application by at least one session of the APPLICATION layer. Each session of this layer serves as a serial access point to the stack: messages are sent to the lower layers by the order they are injected in the access point by the application. Similarly, messages are delivered to the application by the order they are received from the lower layers. A tag is assigned to each descending event by this layer, based on the channel specified by the user. This tag is preserved when the event descends the stack.

The stack interfaces the communication media through a DEVICE layer. This layer is just an abstraction of any protocol outside the control of our communication kernel (for instance, many fault-tolerant stacks interface TCP or UDP transports). The DEVICE layer is responsible for adding to the message a global identifier of the peer channel and for assigning a channel tag to the ascending events (ascending events are triggered by the reception of messages from the device).

### 3.5 Multi-channel sessions

Due to inter-QoS constraints, multiple channels can transverse a single session at one or more layers. These sessions are called *multi-channel sessions*.

When a multi-channel session receives an event it must register the channel on which the event is flowing; if the event is propagated in the stack the associated channel must be used. Consider again our multimedia example. When the intermedia synchronization layer is requested to send an audio frame, it should pass this frame to the audio layer of the audio channel; when a video frame is received it should be passed to the video layer.

Sometimes, multi-channel session may spontaneously generate events which are not bound *a priori* to a given channel. For instance, the intermedia synchronization layer may exchange non-audio and non-video control information with its peers. To avoid ambiguities, every multi-channel session must be associated to a "default" channel that is used to route all events which are not specifically bound to a given channel. In our example, control information could be routed by default through the data channel.

### 3.6 Defining stacks

A stack can be defined as a sequence of channels that may share sessions. Each channel is defined as a sequence of pairs (Layer,Session). Stacks can be easily defined in graphical form, such as illustrated in Figure 5. However, we believe that, for most practical purposes, graphs can be specified using a string with a simple syntax. We propose the syntax presented on Figure 7 for stack definition.

```
Stack="Channel1:
            Layer[Session] |
            Layer[Session]
            ...;
       Channel2:
            ...;
       ...;
       Defaults:
            Layer[Session] -> channel;
            ...;
       "
```

Figure 7: Stack definition syntax

*Channel* is a name that identifies a given path, allowing system to satisfy requirements RQ-1 and RQ-2; *Layer* specifies a layer name and *Session* its instance. Session identifiers are simple integer numbers; note that the session identifier has a scope that is local to the stack specification since, by definition, channels that share a session of a given layer belong to the same stack. With this syntax users can:

- Use different instances of the same protocol for different channels by specifying different session numbers;
- Force channels to share layers by specifying the same session number.

The stack of our first example, graphically represented in Figure 5, would be specified as follows:

```
stack="audio: APPLICATION[1] |
INTERMEDIASYNC[1] | AUDIOPROTO[1] | FIFO[1] |
FAILDETECT[1] | DEVICE[1]; video:
APPLICATION[1] | INTERMEDIASYNC[1] |
VIDEOPROTO[1] | FIFO[2] | FAILDETECT[1] |
DEVICE[1]; data: APPLICATION[1] |
INTERMEDIASYNC[1] | DATAPROTO[1] | FIFO[3] |
FAILDETECT[1] | DEVICE[1]; defaults:
INTERMEDIASYNC[1] -> data"
```

Note that audio and video channels share sessions of the APPLICATION, MEDIASYNC and FAILDETECT layers but have their own local sessions of the FIFO layer.

### 3.7 Execution overhead

At first glance, one might think that the additional flexibility offered by our approach comes at the expense of some execution overhead. At this point, we still do not have a running prototype of our architecture. However, our experience with the Ensemble system lead us to expect a small overhead only during stack creation: the meta-structures required by our approach are more complex than those used in that system. On the other hand, the complexity of event routing should not be seriously affected.

## 4 Related work

The problem of inter-QoS constraints can be addressed using "ad-hoc" solutions. The solution described in [2] for the inter-media synchronization uses a monolithic layer. A monolithic layer is also proposed by Mostefaoui and Raynal [12] to address an ordering problem similar to the one presented earlier. Two classes of messages are supported by a single layer: unordered and causally ordered. Unordered messages can de delivered in any order regarding to each other, but ordered with regard to causal messages.

Horus [13] is a flexible group communication where protocols, in form of layers, are dynamically stacked by user request. Some complex constructs can be found in the architecture: the hierarchical membership layer (PARCLD) offers a reversed cactus stack and the light-weight groups layer (LWG) offers a standard cactus stack. However, these compositions are layer-dependent and are not maintained by the kernel meta-structures (for instance, the LWG layer has a single session).

The Ensemble [5] system follows the Horus project emphasizing the use of formal methods to validate the correctness of protocols. It offers more restrictive layering conditions. Stacks are not allowed to be joined (as in PARCLD layer in Horus) nor to be splited. The Ensemble team has also focused on QoS problems [1]. The Maestro's key concept is to have a "core" stack coordinating others, sharing joins, leaves and failure detectors. Although accomplishing an improvement on resources consumption, Maestro doesn't allow any layer sharing which could optimize the tool even more.

Bast is an object-oriented library of reliable distributed protocols [3]. The use of the Strategy Design Pattern [4] produces a quite intuitive model for the protocol designer and user. Porting our model to Bast would be probably quite easy if requirement RQ-4 could be ignored since Bast's supports only static bindings between protocols.

Consul [11] is a fault tolerant distributed system implemented on the x-Kernel [9]. The system presents interesting features towards our proposal, namely a *divider* layer that performs routing of events based on message classes and allows events to be processed in parallel. However, layers must explicitly notify the divider layer of the classes of messages they are interested in. It is our belief that our approach could be implemented on top of Consul, but with a non-negligible amount of code changes. This work has been extended in [7]. The extended model is based on three components: *micro-protocols*, *events* and *frameworks*. The sys-

tem is event-driven and micro-protocols must register in the framework the set of events they are ready to process. The events may be delivered in parallel or sequentially to the layers. Layers can create new event types. However, the framework does not explicitly manage the notion of session. On the other hand, the work of Hiltunen provides horizontal compound, which we still not address in our architecture.

# 5 Conclusions

This paper presented the case for considering both intra-QoS and inter-QoS constraints when designing communication stacks to support application with multiple QoS requirements. We have presented a model where channels are composed of layer sessions and inter-QoS constraints are modeled by making channels share some sessions of common layers. The model can be implemented in an efficient way by using tag-switching techniques to support the routing of events between adjacent layers.

# References


[1] K. Birman, R. Friedman, and M. Hayden. The maestro group manager: A structuring tool for applications with multiple quality of service requirements. Technical report, Cornell University, Ithaca, USA, February 1997.

[2] M. Correia and P. Pinto. Low-level multimedia synchronization algorithms on broadband networks. In *The Third ACM International Multimedia Conference and Exhibition (MULTIMEDIA '95)*, pages 423–434, San Francisco, November 1995. ACM Press.

[3] B. Garbinato and R. Guerraoui. Flexible protocol composition in Bast. In *Proceedings of the 18th International Conference on Distributed Computing Systems (ICDCS-18)*, pages 22–29, Amsterdam, The Netherlands, May 1998. IEEE Computer Society Press.

[4] Benoît Garbinato and Rachid Guerraoui. Using the strategy design pattern to compose reliable distributed protocols. In USENIX, editor, *The Third USENIX Conference on Object-Oriented Technologies and Systems (COOTS), June 16–19, 1997. Portland, Oregon*, pages 221–232, Berkeley, CA, USA, June 1997. USENIX.

[5] M. Hayden. *The Ensemble System*. PhD thesis, Cornell University, Computer Science Department, 1998.

[6] Mark Hayden and Robbert vanRenesse. Optimizing layered communication protocols. Technical Report TR96-1613, Cornell University, Computer Science, November 18, 1996.

[7] M. Hiltunen and R. Schlichting. An approach to constructing modular fault-tolerant protocols. In *Proceedings of the 12th Symposium on Reliable Distributed Systems*, pages 105–114, Princeton, New Jersey, October 1993. IEEE.

[8] N. Hutchinson and L. Peterson. Design of the x-Kernel. In *Proceedings of the SIGCOMM'88: Communications Architectures and Protocols*, Stanford, USA, August 1988. ACM.

[9] N. Hutchinson and L. Peterson. The x-Kernel: An architecture for implementing network protocols. Technical report, University of Arizona, Tucson, USA, 1990.

[10] S. Mishra, L. Peterson, and R. Schlichting. Consul: A communication substracte for fault-tolerant distributed programs. *Distributed Systems Engineering*, 1(2):87–103, December 1993.

[11] Shivakant Mishra, Larry Peterson, and Richard Schlichting. Modularity in the design and implementation of Consul. Technical Report TR92-20, University of Arizona, Tucson, USA, August 3 1992.

[12] Achour Mostefaoui and Michel Raynal. Definition and implementation of a flexible communication primitive for distributed programming. In *Proceedings of the International Conference on Applications of Parallel and Distributed Computing*. IFIP, April 1994.

[13] Robbert Van Renesse, Kenneth P. Birman, Bradford B. Glade, Katie Guo, Mark Hayden, Takako Hickey, Dalia Malki, Alex Vaysburd, and Werner Vogels. Horus: A flexible group communications system. Technical Report TR95-1500, Cornell University, Computer Science Department, March 23, 1995.

[14] Robbert van Renesse, Kenneth P. Birman, Roy Friedman, Mark Hayden, and David A. Karr. A framework for protocol composition in Horus. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 80–89, Ottawa, Ontario, Canada, 2–23 August 1995.