Optimistic Virtual Synchrony

Jeremy Sussman* Idit Keidar† Keith Marzullo‡

Abstract

Group communication systems are powerful building blocks that facilitate the development of fault-tolerant distributed applications. Such systems generally run in an asynchronous fault-prone environment, and provide semantics (called Virtual Synchrony) that mask the asynchrony and unreliability of the environment. In order to implement Virtual Synchrony semantics, group communication systems typically impose blocking periods during which applications are not allowed to send messages.

This paper presents a novel form of group communication, Optimistic Virtual Synchrony (OVS). OVS allows applications to send messages during periods in which existing group communication services block, by making optimistic assumptions on the network connectivity. OVS allows applications to determine the policy as to when messages sent optimistically should be delivered and when they should be discarded. Thus, OVS gives applications fine-grain control over the specific semantics they require, and does not impose costs for enforcing any semantics that they do not require. At the same time, OVS provides a single easy-to-use interface for all applications. The paper presents several examples of applications that may exploit OVS and empirical results that show the performance benefits of using OVS.

Keywords: Group Communication, Virtual Synchrony, Optimism

*IBM T. J. Watson Research Center. Route 134, Kitchawan Road, Yorktown Heights, NY 10598 Phone: (914)945-2327 E-mail: `jsussman@us.ibm.com`

†MIT Lab for Computer Science. 545 Technology Square, NE43-367, Cambridge, MA 02139, U.S.A. Phone: (617) 253 9302 E-mail: `idish@theory.lcs.mit.edu`

‡University of California, San Diego, Department of Computer Science and Engineering. 9500 Gilman Drive, La Jolla, CA 92093. Phone (858) 534-3729. E-mail: `marzullo@cs.ucsd.edu`

1 Introduction

Group communication systems [1, 36] are powerful building blocks that facilitate the development of fault-tolerant distributed applications. Group communication systems provide the notion of *group abstraction*, which allows processes to be easily organized in multicast groups. Group communication systems typically integrate two types of services: *group membership* and *reliable group multicast*. The membership service maintains a listing of the currently active and connected group members and delivers this information to its clients whenever this information changes. The output of the membership service is called a *view*. Reliable multicast services deliver messages to the current view members. Such communication services complement the membership service.

Group communication systems generally provide some variant of *virtual synchrony* semantics. Virtual synchrony semantics synchronize views with regular messages and thus simulate a "benign" world in which message delivery is reliable within the set of connected processes. Many variants of virtual synchrony semantics have been suggested [30, 20, 36, 13, 33, 19, 25]. Such semantics are especially useful for constructing fault-tolerant applications that maintain consistent replicated state of some sort (e.g., [3, 6, 24, 19, 35, 9, 28]).

The key aspect of virtual synchrony is the interleaving of message send and delivery events with view events. To discuss such interleaving, we associate message send and delivery events with views: we say that an event e occurs at a process p *in view* v if v was the last view delivered to p before e. If no such view was delivered before e, then we say that e occurs in a default initial view v_p. A useful property specified by nearly all variants of virtual synchrony is the agreement among all processes moving together from a view v to another view v' on the set of messages delivered in v. This property has been called *View Synchrony* [36, 11].

All variants of virtual synchrony specify that every message m be delivered in the same view v by all processes that deliver m. This provides a simple but strong consistency property. In addition, many variants (e.g. [12, 20, 30, 25, 23, 19, 16, 17]) strengthen this property to require that the view in which a message is delivered be the same view in which it was sent. This property has been called *group awareness* [34] and *Sending View Delivery* [36]. For the remainder of this paper, we use the term group awareness.

Group awareness is exploited by applications to minimize the amount of context information sent with each message and the amount of computation time needed to process a message. For example, there are cases in which applications only process messages that arrive in the view in which they were sent. This is usually the case with *state transfer* messages sent when new views are delivered (see [4]). By relying on group awareness, such applications need not tag each state transfer message with the view in which it is sent. Group awareness is also useful for applications that send vectors of data corresponding to view members. Such an application can send the vector without annotations, relying on the fact that the ith entry in the vector corresponds to the ith member in the current view (see [20]). Another example of the power of group awareness is illustrated in [35] which identifies a tradeoff between totally ordered multicast and group awareness.

Group awareness is a costly property. Friedman and van Renesse [20] prove that providing group awareness requires that the application be periodically *blocked* from sending messages, or else other useful properties such as View Synchrony (as described above) and Self-delivery (which requires processes to deliver their own messages) cannot be implemented. Therefore, in order to provide group awareness, most group communication systems block processes from sending messages from the time that the need for a view change is recognized until the view is delivered to the application. Such blocking can cause an expensive waste of valuable computation and network resources.

In this paper, we address this waste of resource using an *optimistic* approach. We present *Optimistic Virtual Synchrony (OVS)*, a novel form of group communication that provides the power of group awareness without the performance penalty of blocking. In Optimistic Virtual Synchrony, each view event is preceded by an *optimistic view* event, which provides the application with an estimate of the next view. After this event, applications may optimistically send messages that will provisionally be delivered in the next view. If some application defined property about the next view holds, then the messages will be delivered. Otherwise, the messages are *rolled back*, i.e. they are discarded, and the sending application is informed. Thus, the application specifies the *policy* for optimistic message delivery, and OVS provides the mechanism for implementing any application-specified policy.

We have observed that applications seldom require that the new view be identical to the optimistic one; typical group aware applications are satisfied by weaker constraints. Examples of applications that benefit from OVS appear in Section 3.

We built a version of OVS on top of an existing group communication service, Transis [18]. In its original form, Transis does not provide group awareness to the processes. However, there is a group aware communication mechanism in Transis which is only used internally by the Transis servers. We used this mechanism in implementing OVS on top of Transis. As expected, our performance measurements show that introducing optimism significantly reduces the message delivery latency during view changes. Furthermore, we show that the overhead induced by OVS is very small. We describe the implementation and present our performance measurements in Section 4.

1.1 Evaluating Optimistic Virtual Synchrony

Optimism does not provide additional capabilities for the application programmer. Rather, optimism allows processes to make progress in situations where they would otherwise be forced to block. The utility of optimism can be measured in three ways:

1. By illustrating applications that can make reasonable progress under optimism during periods in which they would otherwise be forced to block. In Section 3, we provide examples of applications that benefit from the optimism provided by OVS.

2. By demonstrating that the additional overhead associated with supporting the optimistic execution is not too great. In Section 4 we measure this by comparing the implementation of OVS on top of Transis with the non-optimistic version of Transis.

3. By demonstrating that the actions performed optimistically are rolled back infrequently enough that the cost of rolling back actions is masked by the gain from optimism. With OVS, the frequency at which optimistic messages will be dropped depends on two factors: the environment and the application-specified policy. An optimistic message is dropped only if new changes of connectivity occur in the environment while a view change is taking place, and these changes are not allowed by the application-defined policy. Modeling the frequency of changes in the environment is not in the scope of this paper. However, we note that since OVS allows the application to specify the message constraint, the fraction of optimistic messages that are dropped is highly application dependent. In fact, in Section 3 we show that there are examples of applications that *never* drop optimistic messages.

   Furthermore, the cost of rolling back the messages is very small, as these messages are sent when the available bandwidth would otherwise not be utilized, and they are merely dropped. In Section 4 we show that a very small amount of computation is needed to determine if an optimistic message is to be delivered or not. Therefore, the cost of rolling back is easily masked by the gain from optimism.

2 The Optimistic Virtual Synchrony Programming Model

Group communication services interact with their applications via an interface consisting of at least three types of events: *send, receive*, and *view*. A *send* event is sent by the application to the group communication service to send a message. A *receive* event is sent by the group communication service to the application to deliver the message. A *view* event is sent by the group communication service to notify the application that the view is changing. A view is a pair, consisting of a set of processes and a unique identifier.

Group aware group communication services require applications to refrain from sending messages while view changes are taking place. To this end, *block* and *flush* events are added to this interface. The group aware service sends a *block* event to the application to inform it that a view change is under way. The application responds with a *flush* event, acknowledging the *block* event. The *flush* event must follow all of the messages sent by the application in the current view. The application then refrains from sending messages until it receives a new *view* from the group communication service. Such services use the blocking mechanism to ensure group awareness, i.e., that every message is delivered in the view in which it was sent (see [20, 25]).

With OVS, the *block* event is replaced by an optimistic view event, *optView* which contains a set of members. This set is an estimate of what the set of members in the next view will be. Group membership algorithms (e.g., [26, 5]) can usually provide an optimistic view which is accurate unless further changes in the system connectivity occur during the view change. When the application receives the *optView* event, it sends a *flush* event and enters *optimistic mode*. In this mode, the application still receives messages that were sent in the view that it is leaving, but at the same time, the application may optimistically send messages to be provisionally delivered in the next view. The messages sent in optimistic mode are called *optimistic messages*. When the group communication service delivers a new *view* to the application, the application returns to *normal mode* and sends a *viewAck* event to the group communication service to denote the end of the optimistic mode. In the normal mode, the application sends regular messages to be delivered in the same view. This program flow is depicted in Figure 1.

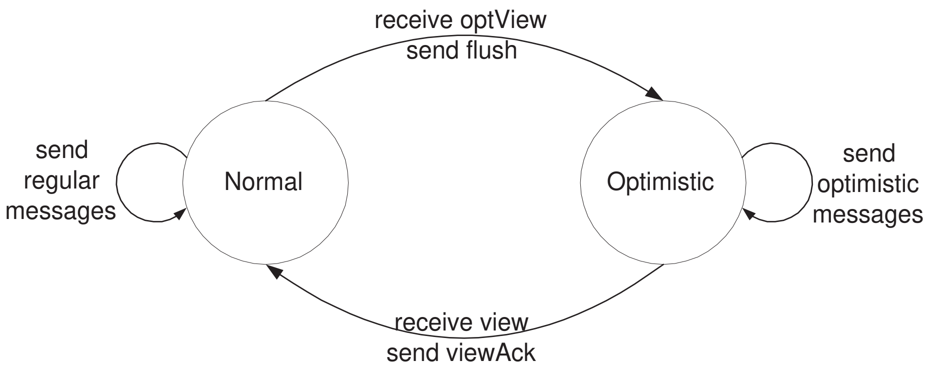


Figure 1: Process modes in Optimistic Virtual Synchrony.

When the new view is delivered, the group communication service checks whether the optimistic messages should be delivered in the new view or not. This is checked by applying an application-provided predicate, *MessageCondition*, to each optimistic message. If the predicate is evaluated to *true*, the message is delivered. Otherwise, the message is discarded at all locations except for the sender. The sender is informed of any non-delivered optimistic messages via the *discardedMessages* event.

The parameters of the *MessageCondition* predicate include the set of members of the new view and the optimistic view in which the message was sent, as well as the message for which the condition is being checked. Group communication services that supplement views with information regarding previous views of other members, (e.g., [15, 10], or the transitional view/set of [30, 4, 36, 25]) can provide this information as a parameter to the predicate as well. Examples of *MessageCondition* predicates are discussed in Section 3 below.

Note that, in particular, in order to evaluate the *MessageCondition* predicate, each process needs to learn of every other view members' optimistic views. This does not require additional messages to be sent, but rather the optimistic view can be piggy-backed on the messages necessary for the agreement on the next view. The OVS group communication service can provide the optimistic view information to any application that is interested in it. We give an example of such an application in Section 3.4.

3 Example Applications of Optimistic Virtual Synchrony

In this section we present several different applications that benefit from Optimistic Virtual Synchrony. These applications are meant to be illustrative of the power of OVS, but are by no means exhaustive. Another example of an application that exploits OVS is the *Bancomat* resource allocation algorithm of [35] (see [34] for details). In addition to these examples, OVS can be used by applications that do not require group awareness (e.g., [14, 3, 9]) by always evaluating the *MessageCondition* predicate to *true*.

3.1 Primary Views

Applications that maintain globally consistent shared state (for example, [21, 24, 6, 19, 28, 32, 22, 27, 2]) usually avoid inconsistencies by allowing only members of one view (the *primary view*) to update the shared state at a given time. Different primary views can be defined for different replicated objects. Such applications use group awareness: messages that update an object are sent only in this object's primary view, whereas query messages are sent in all views.

Consider, for example, an application in which each object has a designated master site such that the object is updated only in a view containing that site. The optimistic *MessageCondition* predicate for such an application might be:

```
boolean MessageCondition( set newView, optView, char *m )
   return ( m.type = query ∨ masterCopyOf(m.object) ∈ newView )
```

Likewise, if there is no designated master site, then the predicate can check if the new view contains a majority (or quorum) of $m.object$'s copies. When a message is rolled back, the sender stores the request until the view changes to a primary one.

Note that the format of a message m can be used in the *MessageCondition* predicate although the OVS service does not know this format. This is one benefit of the application specifying the predicate.

3.2 State Transfer

Typical applications of group communication services, (e.g., [4, 35, 23, 21, 4, 2, 24, 6, 28, 37]), engage in state transfer whenever a new view is delivered. State transfer messages are usually utilized only if they are *fresh*, i.e., they pertain to the current view. Therefore, applications that send state transfer messages usually require group awareness, or impose group awareness by tagging each state transfer message with the view in which it was sent and discarding messages pertaining to old views (see [4]).

Note that state transfer messages cannot be sent optimistically for all applications. While the application is in optimistic mode, messages from the previous view may arrive. If such a message can cause the application to change its state, then a state transfer message sent optimistically before the message arrives will not reflect the updated state. However, we identify two cases in which state transfer messages may be sent optimistically:

1. When the application state is too large to be sent in a single message, for example, when a replica is being added in a replicated file (or database) server (e.g., [24, 21, 6, 2]). Using OVS, the application can begin to send the state while in optimistic mode, and send only the last part of the state (reflecting the latest changes) when the new view is delivered.

2. When the application state changes only following view changes. For example, in a service that uses dynamic voting to determine the primary view [37], when a new view is delivered all of the applications exchange information about past primary views and use the state transfer messages to determine the state of the current view. This state does not undergo further changes during the same view. Thus, the state can be sent in optimistic mode.

Note that the applications identified above do not need any guarantees about the ensuing view for the state transfer message to be correct. Thus, the *MessageCondition* predicate for state transfer messages is always evaluated to *true*. The only guarantee needed is a fresh delivery property: that the message will be delivered in the next view, not in some view further in the future. This points out one of the strengths of OVS: group awareness as specified in [20, 30, 19, 23, 16, 17, 25] and provided by group communication services such as [12, 7, 20] provides a much more costly abstraction than is needed for this application. On the other hand, group communication services such as [18] that are not group aware do not provide this fresh delivery property.

3.3 Waiting for State Transfer to Complete

Many applications that exchange state transfer messages when a new view is delivered (e.g., [21, 24, 6, 19, 28, 2]) refrain from sending messages until all state transfer messages are received. Thus such applications extend the blocking period imposed by a group aware service until the state transfer is complete. However, many applications need not engage in state transfer upon receipt of every new view. Several group communication services provide applications with a set of processes that are known to have retained agreement on the sequence of delivered views. Such a set is called the *transitional view/set* in [30, 4, 36, 25]. If the transitional set is a superset of the new view, then such applications need not engage in state transfer (see [4, 36]).

Such applications can benefit from OVS by sending messages optimistically, and delivering these messages only if the new view is a subset of the transitional set, i.e., if no state transfer is necessary. The *MessageCondition* predicate for such an application might be:

```
boolean MessageCondition( set newView, optView, char *m, set transitional )
   return ( newView ⊆ transitional )
```

If the optimistic assumption is false and state transfer is needed, the messages sent optimistically will be rolled back, and the application can re-send the information after the state transfer has been completed.

3.4 Data Vectors

Group awareness is useful for applications that send vectors of data corresponding to processes: group awareness allows such applications to send the vector without annotations, relying on the fact that the ith entry in the vector corresponds to the ith member in the current view. This reduces the amount of context information sent with each message and the amount of computation time for processing messages (see [20]).

Using OVS, such applications may also send optimistic messages containing data without annotations while they are in the optimistic mode. When the view is delivered, the application may request the OVS service for the optimistic views of all of the view members. These can be used to create conversion tables which convert an index in each sender's optimistic view to a corresponding index in the new view, and to remove entries in the vector which correspond to members that are not in the new view. The *MessageCondition* predicate in this case is always evaluated to *true*.

This technique induces some processing overhead, but only on the processing of optimistic messages. In normal mode, the application can continue to benefit from group awareness with no additional overhead.

3.5 Causal Multicast

An example of an application that sends vectors of data corresponding to processes is an implementation of causal multicast [29] using *vector clocks*. Causal multicast ensures that by the time a process p receives a multicast message m sent by a process q, p has also received all of the messages that q received before sending m. A vector clock is a vector of integers, indexed by the set of processes in the system. The value of the vector clock of p for some process q represents the sequence number of the last message multicast by q that p has received. When a message m is multicast by a process q, m includes a copy of the vector clock at q. If p receives m from q and the vector clock value in m for some other process s is greater than the vector clock value of p for s, then p knows that there is a message from s that causally precedes m that p has not yet received and p cannot deliver m yet (see [29]).

This technique is used for implementing causal group multicast in the ISIS and Horus group communication systems. In a group based programming environment, the overhead associated with causal multicast can be greatly reduced. In each view, view members receive messages only from other members of the same view. Therefore, if the processes in the new view agree on the messages received before the view change, then only the vector clock values for the processes in the view need to be included in further messages. A group aware environment nicely supports this implementation.

As explained in Section 3.4 above, using OVS the application can continue to send vectors without annotations while in the optimistic mode. However, in order to preserve causality, the vector has to include indices corresponding to *all* the members of the view. Therefore, optimistic messages sent with a partial vector that does not include all of the members of the new view should be discarded.

This implementation of causal multicast was a main influence on the design of the *Weak Virtual Synchrony (WVS)* programming model of Horus [20]. When a view change is taking place, WVS provides applications with *suggested views* and guarantees that the ensuing view will be an ordered superset of the suggested view. Processes may send messages during the suggested view, and these messages will be delivered in the ensuing view (see Section 5). Friedman and van Renesse exploit Weak Virtual Synchrony for causal vectors as follows: in suggested views, processes send vectors which include entries for all of the members of the suggested view. When the message is delivered in the ensuing view, the entries in the vector pertaining to processes that left the view are filtered out. OVS can be used in the same manner: processes can send vectors vectors which include entries for all of the members of the optimistic view. If the ensuing view is indeed a superset of the optimistic view, the messages can be processed as with Weak Virtual Synchrony. Otherwise, they should be rolled back. The message condition is as follows:

```
boolean MessageCondition( set newView, optView, char *m )
   return ( newView / optView ≠ {} )
```

When a message is rolled back, the sender re-sends the information with the appropriate vector clock. We further compare WVS with OVS in Section 5.

4 Implementation and Performance Results

We have implemented Optimistic Virtual Synchrony on top of the Transis group communication service [18]. We chose Transis because it has two different modes of group communication: it provides the application processes with non-blocking delivery which does not provide group aware semantics, and it internally uses blocking delivery which does provide group aware semantics. We had two goals for this implementation:

1. To understand what extra support would be needed to provide OVS on top of an existing group aware group communication service.

2. To compare the performance of OVS with that of a group aware group communication service as well as with the performance of a non-group aware one. By implementing OVS on top of the two versions of Transis, we could better ensure that such a comparison be fair.

In Section 4.1 we describe how we met the first goal. The second goal is discussed in Section 4.2.

4.1 Implementing Optimistic Virtual Synchrony in Transis

The Transis group communication service is structured around a group of servers. The Transis servers communicate with each other using reliable FIFO links. When the need for a view change is recognized by some server, this server sends synchronization messages to the other servers to denote the end of the current view. If a server receives a synchronization message without having detected the need for a view change itself, it treats the synchronization message as a detection and also engages in the view change algorithm. Each server refrains from sending new messages after sending the synchronization message and until the new view is delivered. Thus, group awareness is supported among the Transis servers.

With OVS, optimistic messages can be sent during this time period. The synchronization messages together with the FIFO order guarantee that when an optimistic message reaches a Transis server, this server is either also in the optimistic mode for the same view, or in the subsequent regular view, as illustrated in Figure 2.

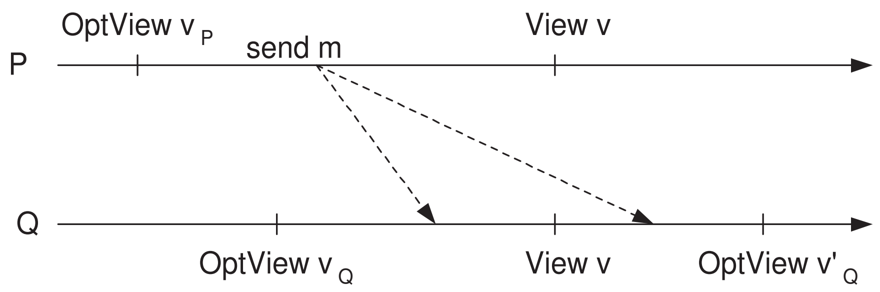


Figure 2: Possible arrival times of optimistic messages at Transis servers.

In order to implement OVS in Transis, we added code in the following places:

1. <u>When the need for a view change is recognized by a Transis server.</u> An *optView* is sent to the application processes. Once an application responds with a *flush*, it enters optimistic mode (see Figure 1 above). The mode of the application (optimistic or normal) is saved in the OVS process.

2. <u>When a message is sent by a process.</u> If the sending application is in optimistic mode then the message is marked as optimistic before sending it to the members of the optimistic view.

3. <u>When a message is received by a process</u> (including loop-back receipt of a message by its sender). If the message is not marked as optimistic, it is not handled by OVS code but is rather passed to the regular Transis code as usual. Otherwise, there are two cases handled differently in OVS:

(a) If a view change is under way, then the optimistic message is enqueued in a buffer for optimistic messages, and its receipt is masked from the Transis code.

(b) If a view change is not under way, then, as explained above, the optimistic message must have been sent during the optimistic mode preceding the current view. In this case, the *MessageCondition* is applied to the message in order to determine whether the message should be delivered or not.

4. When a new view is delivered to a process by Transis. Each message in the optimistic message buffer is checked to see if the sender is a member of the new view, and then the *MessageCondition* for the message is checked. Depending on the result, the message is either delivered or dropped, and the process is notified via a *discardedMessages* event that a message that it sent will not be delivered.

   In addition, if the new view contains members which are not members of the optimistic view, then the optimistic messages will not have been sent to these new view members. These messages are forwarded to the new members. However, a selective sending mechanism does not exist in Transis. Instead, we used the Transis retransmission mechanism which re-sends the message to the entire view. Those that had not previously received the message would therefore receive it, and those that had previously received it simply ignore it.

   When the application responds with a *viewAck*, its mode is changed to normal in the OVS code.

4.2 Performance Measurements

In this section we describe the measured performance of OVS implemented on top of Transis. The tests described below were run on three Sun UltraSparc 5/10s, each of which was running at 333 Megahertz. All three machines were running SunOS version 5.6. The three machines were connected via 100MBit/sec Ethernet. The machines were not being used by any other users during these tests. In each test, no fewer than 40,000 messages were sent. All messages in the tests were about 1 kilobyte long, a batch of 15 messages was sent every 15 milliseconds. Each test was repeated at least three times to ensure that the results were not spurious.

We sought to measure two different aspects of OVS. The first measurement was of the overhead associated with processing of messages in OVS. The second measurement was a comparison of the average time to deliver messages after a view change in OVS versus the average time to deliver messages after a view change in Transis. We describe the two measurements in greater detail below.

4.2.1 The overhead of OVS

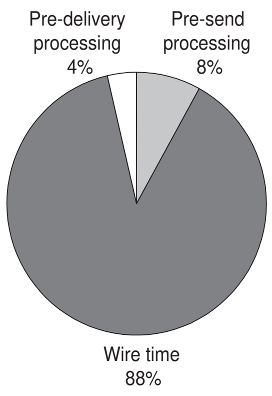
The life-cycle of a message in Transis can be roughly described as consisting of three stages: *pre-send processing*, *wire time* and *pre-delivery processing*. When a message is sent by an application process in Transis, the Transis process associated with the sender performs some pre-send processing (e.g., marshaling of header information) before sending it on the communication stratum or handing it off to its own reception handler. When a message is received by a Transis server, pre-delivery processing (e.g., demarshaling, ensuring that it meets delivery semantics) is performed. When this processing has been completed, the message can be delivered to the application process.

As expected, our measurements show that the wire time is the most significant component in the message life-cycle. This is illustrated in Figure 3 (a): The average pre-send processing time was consistently around 90 microseconds for all of the tests we ran. The average wire time was around 1000 microseconds, and the average pre-delivery processing time on the server side was consistently around 40 microseconds for all of the tests.

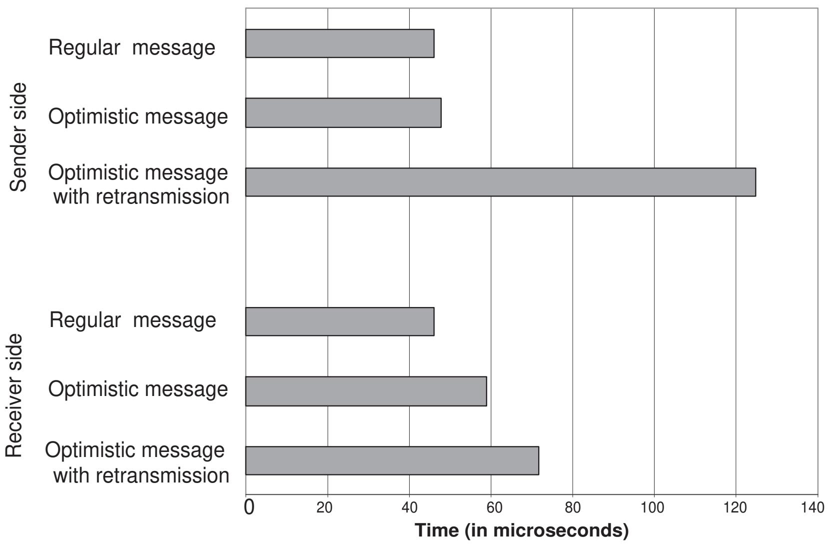
The main performance gain of using OVS instead of a group aware group communication service is the masking of the pre-send processing time and wire time for optimistic messages. This results from messages being sent during the time that a group aware group communication service would block. In our experiments, we wanted to demonstrate that this performance gain is significantly larger than any overhead induced by OVS in the pre-delivery processing time[1]. We compared the pre-delivery processing time of OVS to that of regular messages in Transis. The results are shown in Figure 3 (b).

When measuring the pre-delivery processing time of OVS, we distinguish between two cases: (1) the new view contains only members that were in the optimistic view, hence no message retransmission (or forwarding) is needed; and (2) the new view does contain new members and retransmission is needed. In the latter case the pre-delivery processing time is larger since it includes the time required to retransmit the message. We measured the pre-delivery processing time both at the sender side and at the receiver side. As expected, when no retransmissions were needed the processing time was only slightly larger with OVS: around 50 microseconds at the sender side and around 60 at the receiver. When retransmissions are necessary, the sender side pre-delivery processing time grows to almost 125 microseconds per message, and the receiver side grows to slightly over 70 microseconds per message. Retransmissions slow down the sender which has to engage in retransmitting them; they also

[1] During the pre-send processing, OVS only adds one bit of information to the message header denoting that it is an optimistic one, therefore, the overhead OVS induces on the pre-send processing time is negligible, and the overhead of using OVS affects mainly the pre-delivery processing time.



(a) Message life-cycle in Transis.



(b) Transis and OVS pre-delivery processing times.

Figure 3: Transis processing times.

slow down the receiver since in Transis messages are retransmitted to all of the processes, and therefore the receiver receives duplicates of these messages.

All in all, we observe that the overhead induced by OVS is smaller by an order of magnitude that the performance gain from masking the pre-send processing and the wire times.

To further understand the overhead associated with the optimistic message processing, we divided the pre-delivery processing into time spent evaluating the message condition and time spent iterating over the optimistic message buffer and handling the messages. In Figure 4 we show this breakdown. In all cases, about 8 microseconds per message are spent in the message condition evaluation. We experimented with several simple message conditions, and there was no measured difference. The remaining difference between the pre-delivery processing of optimistic and non-optimistic messages, which is quite small on the server side and a bit larger on the receiver side, can be attributed to overhead in iterating over the buffer and changing the fields of the message where necessary. In the case of retransmission, most of the overhead on the sender side is in the actual retransmission of the message.

4.2.2 Latency of optimistic messages

The direct benefit of Optimistic Virtual Synchrony over a group aware group communication service is that OVS allows messages to be sent during the view change while still providing group aware semantics. Transis normally provides applications with non-group aware semantics, which allows applications to send messages during view changes. In this case the messages are buffered by the Transis server at the sender side during the view change, as

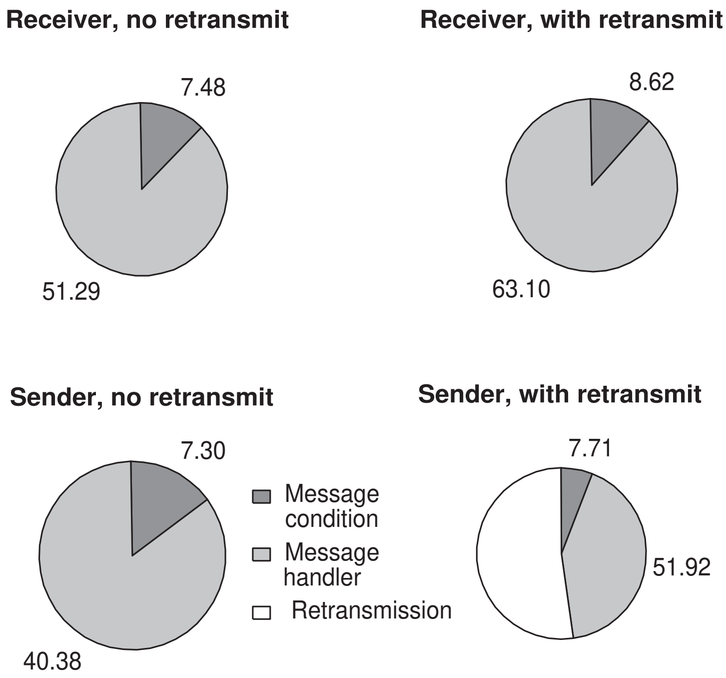


Figure 4: Breakdown of pre-delivery processing time for optimistic messages (microseconds).

opposed to being buffered at the receiver side with OVS and not being sent at all with a group aware service. Thus, the latency associated with the communication should be masked by OVS. We sought to measure this benefit.

To do so, we formulated the following function f: Consider a run in which a single process p sends a stream of messages, and during which the view changes exactly once. Define $deliver_q(m)$ to be the time that a message m of this stream is delivered by a process q. Let m_0 be the last message sent by p before p is notified that a view change is beginning. Number the messages following m_0 as m_1, m_2, etc. Thus, m_1 is the first message that is affected by the view change: i.e., in OVS, it is the first optimistic message; in the group aware Transis it is the first message sent after blocking; and in the normal Transis, it is the first message buffered during the view change. Now, we define the function f as:

$$f(q,k) \stackrel{\text{def}}{=} (deliver_q(m_k) - deliver_q(m_0))/k$$

Function f gives the average time it takes to deliver a message at a process after a view change has begun. The same function, outside of a view change, would be the inverse of the throughput of the system. That is, f is the frequency with which messages are delivered.

We measured the values of f for four different communication modes:

1. the regular non-group aware mode provided by Transis;

2. a group aware version of Transis;

3. Optimistic Virtual Synchrony without the need for retransmission; and

4. Optimistic Virtual Synchrony with the need for retransmission.

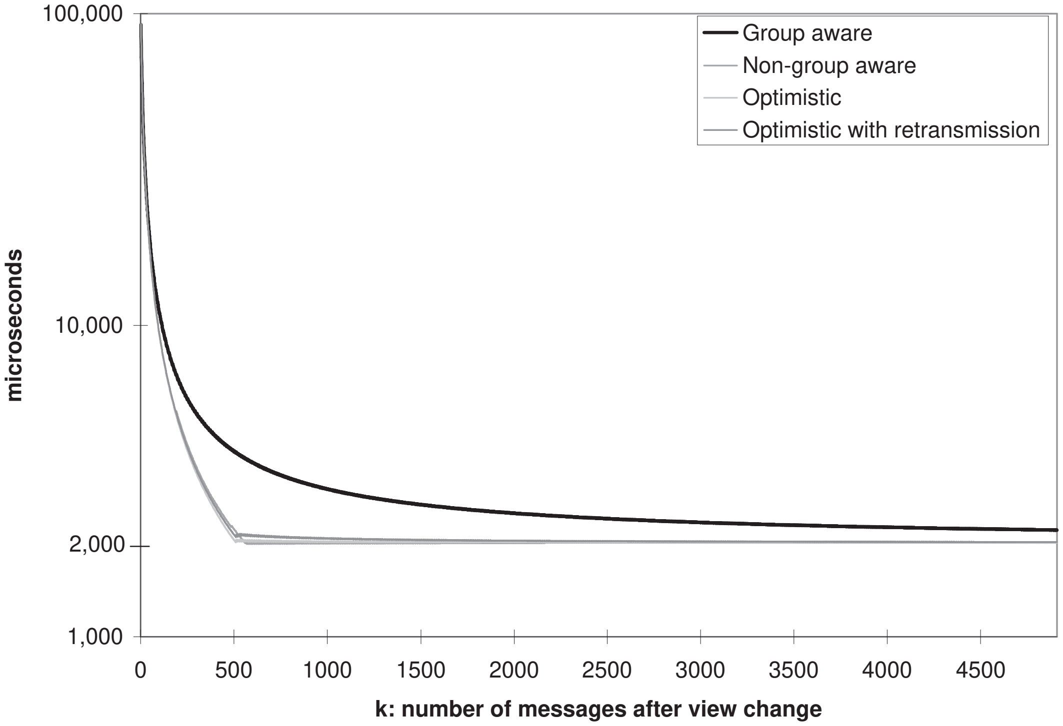


Figure 5: $f(p, k)$ for sender delivery (in logarithmic scale).

Figure 5 shows the measured $f(p, k)$ in logarithmic scale for these four communication modes for one of the tests. Since p is the sender in these tests, this graph shows the function for the local delivery of messages and is not affected by the communication latency. The receiver side for this test showed similar behavior, as did the other tests that were run. Due to the scale of these measurements relative to the differences in the values, the three modes other than the group aware version of Transis cannot be seen separately. For further details, see [34].

We observe that although OVS can provide an application with group awareness, it still allows for communication speeds comparable to the non-group aware mode of Transis, and significantly superior to those of the group aware version of Transis.

For completeness, we also compared the average time to deliver messages outside of view changes in Transis with the average time to deliver non-optimistic messages in the version of Transis which supports OVS. As expected, these times were equal, i.e., OVS induces no delay on the delivery of non-optimistic messages.

5 Related Work

We are not the first to consider the use of optimism to support group communication. In [31], optimistic assumptions are made about the order in which messages are received in order to quickly provide total ordering on the message delivery. In our approach, optimistic assumptions are made about the view, in order to allow message sending during periods of instability. The optimism of [31] is orthogonal to our use, and the two approaches could be combined.

Optimistic Virtual Synchrony allows applications to send messages during periods in which group aware group communication systems block. Two other (non-optimistic) approaches to eliminate the blocking imposed by group aware group communication services have been suggested: *light-weight groups* and Weak Virtual Synchrony.

Light-weight groups are used in systems that are built around a small number of servers that provide group communication services to numerous application clients (for example Transis [18] and Spread [8]). In these systems, client membership is implemented as a light-weight layer that communicates with a heavy-weight group aware layer asynchronously using a FIFO buffer. The asynchrony may cause messages to arrive in later views than the ones in which they were sent. However, since the asynchronous buffer preserves the order of *receive* and *view* events, messages are delivered in the same view at all destinations. The semantics provided by light-weight group membership services, which are not group aware, are too weak for many applications as illustrated in Section 3 above.

In order to eliminate the need for blocking while still providing support for a certain type of group aware applications, Friedman and van Renesse [20] introduce the Weak Virtual Synchrony *(WVS)* programming model. In WVS, every view v is preceded by at least one *suggested view* event. The membership of the suggested view is guaranteed to be an ordered superset of v. Group awareness is replaced by the weaker requirement that every message sent in the suggested view is delivered in the next regular view. This allows processes to send messages while the view change is taking place. The processes that use WVS maintain translation tables that map process ranks in the suggested view to process ranks in the new view. Thus, although messages are no longer guaranteed to be delivered in the view in which they were sent, an application may still send vectors of data corresponding to processes without annotations.

One shortcoming of WVS is that it is useful only for group aware applications that are satisfied with knowledge of a superset of the actual view, and does not suffice for other group aware applications that have different requirements about the ensuing view (see examples in Section 3 above). In contrast, OVS provides applications with the flexibility to determine the policy as to what the ensuing view must be for the messages to be processed. In particular, applications designed to work with WVS can exploit OVS by requiring the ensuing view to be a subset or the optimistic view (see Section 3.4 above).

A second shortcoming of the WVS model is that once a suggested view is delivered, new processes are not allowed to join the next regular view. If a new process joins while a view change is taking place, a protocol implementing WVS is forced to deliver an *obsolete* view, and then immediately start a new view change to add the joiner. Furthermore, WVS requires processes that continue together to the same new view to deliver each other's suggested views. Therefore, if two connected processes deliver conflicting suggested views, then they are forced to deliver views excluding each other before they can deliver a common view again. This imposes severe limitations on the membership service's choice of the next view and forces the membership service to deliver obsolete views. In contrast, OVS does not impose any limitations on the membership service's choice of the next view, hence OVS does not require the membership service to deliver obsolete views. We believe that obsolete views should be avoided since they cause extra overhead for applications to process and increase network congestion by withholding information from applications that might allow them to avoid sending messages that will be discarded (see [26]).

6 Conclusions

We have presented Optimistic Virtual Synchrony, a novel form of group communication which provides the power of group awareness without the execution penalty of blocking.

Optimistic Virtual Synchrony provides applications with the flexibility to determine the policy (message condition) as to when optimistic messages should be delivered and when they should be discarded (rolled back). We have described several different applications that can benefit from OVS. Our examples illustrate how different applications can use OVS with different message conditions. In particular, we have observed that applications seldom require that the new view be identical to the optimistic one; typical group aware applications are in fact satisfied by weaker constraints. We believe that the flexibility to specify the message condition is important, as it gives applications fine-grain control over the specific semantics they require, and does not impose costs for enforcing any semantics that they do not require.

At the same time, OVS provides a single easy-to-use interface suitable for all applications.

We have shown that the overhead induced by OVS on the processing of optimistic messages is smaller by an order of magnitude than the performance benefit gained from sending messages while a group aware service would block. We have shown that the latency of optimistic messages sent using OVS is similar to the latency of messages sent during view changes using a non-group aware group communication service. This latency is significantly smaller than the latency imposed by the blocking period in a group aware service.

Acknowledgments

We thank Roger Khazan and Danny Dolev for many interesting discussions.

References

[1] ACM. *Commun. ACM 39(4), special issue on Group Comm. Systems*, April 1996.

[2] O. Amir, Y. Amir, and D. Dolev. A highly available application in the Transis environment. In *Proceedings of the Hardware and Software Architectures for Fault Tolerance Workshop, at Le Mont Saint-Michel, France*, June 1993. LNCS 774.

[3] Y. Amir, D. Breitgand, G. Chockler, and D. Dolev. Group communication as an infrastructure for distributed system management. In *3rd Intl. Workshop on Services in Dist. and Networked Environment (SDNE)*, pp. 84–91, June 1996.

[4] Y. Amir, G. V. Chokler, D. Dolev, and R. Vitenberg. Efficient state transfer in partitionable environments. In *2nd European Research Seminar on Advances in Dist. Syst. (ERSADS'97)*, pp. 183–192. BROADCAST (ESPRIT WG 22455), Operating Systems Laboratory, Swiss Federal Institute of Technology, Lausanne, March 1997.

[5] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Membership algorithms for multicast communication groups. In *6th Intl. Workshop on Dist. Algorithms (WDAG)*, pp. 292–312. Nov. 1992.

[6] Y. Amir, D. Dolev, P. M. Melliar-Smith, and L. E. Moser. Robust and Efficient Replication using Group Communication. Technical Report CS94-20, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1994.

[7] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. The Totem single-ring ordering and membership protocol. *ACM Trans. Comput. Syst.*, 13(4), Nov. 1995.

[8] Y. Amir and J. Stanton. The Spread Wide Area Group Communication System. TR CNDS-98-4, Johns Hopkins University, 1998.

[9] T. Anker, D. Dolev, and I. Keidar. Fault tolerant video-on-demand services. In *19th Intl. Conf. on Dist. Comp. Syst. (ICDCS)*, pp. 244–252, June 1999.

[10] Ö. Babaoğlu, A. Bartoli, and G. Dini. On programming with view synchrony. In *16th Intl. Conf. on Dist. Comp. Syst. (ICDCS)*, pp. 3–10, May 1996.

[11] Ö. Babaoğlu, R. Davoli, and A. Montresor. Failure Detectors, Group Membership and View-Synchronous Communication in Partitionable Asynchronous Systems. TR UBLCS-95-18, Department of Conmputer Science, University of Bologna, Nov. 1995.

[12] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *11th ACM SIGOPS Symp. on Operating Syst. Prin. (SOSP)*, pp. 123–138. ACM, Nov 1987.

[13] K. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, 1994.

[14] G. Chockler, N. Huleihel, and D. Dolev. An adaptive totally ordered multicast protocol that tolerates partitions. In *17th ACM Symp. on Prin. of Dist. Comp. (PODC)*, pp. 237–246, June 1998.

[15] F. Cristian and F. Schmuck. Agreeing on Process Group Membership in Asynchronous distributed systems. Technical Report CSE95-428, Department of Computer Science and Engineering, University of California, San Diego, 1995.

[16] R. De Prisco, A. Fekete, N. Lynch, and A. Shvartsman. A dynamic view-oriented group communication service. In *17th ACM Symp. on Prin. of Dist. Comp. (PODC)*, pp. 227–236, June 1998.

[17] R. De Prisco, A. Fekete, N. Lynch, and A. Shvartsman. A dynamic primary configuration group communication service. In *13th Intl. Symp. on Dist. Comp. (DISC)*, pp. 64–78, Bratislava, Slovak Republic, 1999.

[18] D. Dolev and D. Malkhi. The Transis approach to high availability cluster communication. *Commun. ACM*, 39(4), April 1996.

[19] A. Fekete, N. Lynch, and A. Shvartsman. Specifying and using a partionable group communication service. In *16th ACM Symp. on Prin. of Dist. Comp. (PODC)*, pp. 53–62, Aug. 1997.

[20] R. Friedman and R. van Renesse. Strong and Weak Virtual Synchrony in Horus. TR 95-1537, dept. of Computer Science, Cornell University, Aug. 1995.

[21] R. Friedman and A. Vaysburg. Fast replicated state machines over partitionable networks. In *16th IEEE Intl. Symp. on Reliable Dist. Syst. (SRDS)*, October 1997.

[22] R. Guerraoui and A. Schiper. Transaction model vs virtual synchrony model: bridging the gap. In *Theory and Practice in Dist. Syst.*, LNCS 938, pp. 121–132. Sept. 1995.

[23] M. Hiltunen and R. Schlichting. Properties of membership services. In *2nd Intl. Symp. on Autonomous Decentralized Syst.*, pp. 200–207, 1995.

[24] I. Keidar and D. Dolev. Efficient message ordering in dynamic networks. In *15th ACM Symp. on Prin. of Dist. Comp. (PODC)*, pp. 68–76, May 1996.

[25] I. Keidar and R. Khazan. A client-server approach to virtually synchronous group multicast: Specifications and algorithms. Tech. report, MIT Lab. for Comp. Sci., 1999. In preparation.

[26] I. Keidar, J. Sussman, K. Marzullo, and D. Dolev. A Client-Server Oriented Algorithm for Virtually Synchronous Group Membership in WANs. Technical Report CS99-623, Department of Conmputer Science and Engineering, University of California, San Diego, June 1999. Also MIT Technical Memorandum MIT-LCS-TM-593.

[27] B. Kemme and G. Alonso. A suite of database replication protocols based on group communication primitives. In *18th Intl. Conf. on Dist. Comp. Syst. (ICDCS)*, May 1998.

[28] R. Khazan, A. Fekete, and N. Lynch. Multicast group communication as a base for a load-balancing replicated data service. In *12th Intl. Symp. on Dist. Comp. (DISC)*, pp. 258–272, Andros, Greece, Sep. 1998.

[29] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 78.

[30] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended virtual synchrony. In *14th Intl. Conf. on Dist. Comp. Syst. (ICDCS)*, pp. 56–65, June 1994.

[31] F. Pedone and A. Schiper. Optimistic atomic broadcast. In *12th Intl. Symp. on Dist. Comp. (DISC)*, pp. 318–332, Sep. 1998.

[32] A. Schiper and M. Raynal. From group communication to transactions in distributed systems. *Communications of the ACM*, 39(4):84–87, Apr. 1996.

[33] A. Schiper and A. Ricciardi. Virtually synchronous communication based on a weak failure suspector. *Digest of Papers, FTCS-23*, pp. 534–543, June 93.

[34] J. Sussman. *Group Communication Services versus Wide-Area Networks*. PhD thesis, University of California, San Diego, 1999.

[35] J. Sussman and K. Marzullo. The *Bancomat* problem: An example of resource allocation in a partitionable asynchronous system. In *12th Intl. Symp. on Dist. Comp. (DISC)*, Sep. 1998.

[36] R. Vitenberg, I. Keidar, G. V. Chockler, and D. Dolev. Group Communication Specifications: A Comprehensive Study. Tech. report CS99-31, Comp. Sci. Inst., The Hebrew University of Jerusalem and MIT Technical Report MIT-LCS-TR-790, Sep. 1999.

[37] E. Yeger Lotem, I. Keidar, and D. Dolev. Dynamic voting for consistent primary components. In *16th ACM Symp. on Prin. of Dist. Comp. (PODC)*, pp. 63–71, Aug. 1997.