Optimistic Virtual Synchrony*

Jeremy Sussman
IBM T. J. Watson Research Center
30 Saw Mill River Road, Hawthorne, NY 10532
jsussman@us.ibm.com

Idit Keidar
MIT Lab for Computer Science
545 Tech. Square, Cambridge, MA 02139
idish@theory.lcs.mit.edu

Keith Marzullo
University of California, San Diego
Department of Computer Science and Engineering
9500 Gilman Drive, La Jolla, CA 92093
marzullo@cs.ucsd.edu

Abstract

We present Optimistic Virtual Synchrony (OVS), a new form of group communication which provides the same capabilities as Virtual Synchrony at better performance. It does so by allowing applications to send messages during periods in which services implementing Virtual Synchrony block. OVS also 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.

1. Introduction

Group communication systems [1, 26] are powerful building blocks that facilitate the development of fault-tolerant distributed applications; they provide an abstraction which allows processes to be easily organized into multicast groups. Group communication systems typically integrate two types of services: *group membership* and *reliable group multicast*. The group membership service maintains a listing of the currently active and connected group members and delivers this information to its clients whenever the list changes. The output of the membership service is called a *view*. The reliable group multicast service delivers messages to the current view members.

Group communication systems generally provide some variant of *Virtual Synchrony* semantics (e.g., [21, 14, 26, 24, 12, 18]), that mask environment failures, and simulate a "benign" world in which message delivery is reliable within each view. Such semantics are especially useful for fault-tolerant applications that maintain some consistent replicated state (e.g., [3, 17, 12, 25, 6, 20]).

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. A useful property specified by nearly all variants of Virtual Synchrony is that processes moving together from a view v to another view v' deliver the same set of messages in v. This property has been called *View Synchrony* or, by itself, *Virtual Synchrony* (see [26]).

All variants of Virtual Synchrony specify that every message m be delivered in the same view v by all processes that deliver m. Many variants (e.g., [9, 14, 21, 18, 16, 12, 24]) strengthen this property to require that the view in which a message is delivered be the same view in which it was sent. This latter property has been called *Sending View Delivery* (see [26]).

Sending View Delivery is exploited by applications to reduce the amount of context information sent with each message. Processes that are in the same view share a common state that is derived from the messages that they have jointly received in that view. Sending

*This work supported in part by UT Austin grant 97-0039 (subcontract to DARPA), Air Force Aerospace Research (OSR) grant F49620-00-1-0097, Nippon Telegraph and Telephone (NTT) grant MIT9904-12, and by NSF grants CCR-9909114 and EIA-9901592.

View Delivery allows a process that receives a message to easily deduce information about the state of another process when that process sent the message. For example, Sending View Delivery has been used for applications that send vectors of data corresponding to view members. Such an application can rely on the fact that the ith entry in the vector corresponds to the ith member in the current view. Hence, the identities of the members need not be included in the message (see [14]). Another application that can use Sending View Delivery is state transfer (cf. Section 3.2).

Providing Sending View Delivery is costly: it requires that the application be periodically blocked from sending messages. Otherwise, a process that persists in sending messages during a view change might indefinitely postpone the view change (assuming that other important properties such as View Synchrony and Self-delivery—a process delivers its own messages—are provided). Therefore, in order to provide Sending View Delivery, 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 wastes valuable computation and network resources.

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 Sending View Delivery without the performance penalty of blocking. In OVS, 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 conditionally be delivered in the next view. Messages are only sent but not delivered optimistically; thus OVS never causes applications to roll back their states.

Once the next view is delivered, the OVS service checks if some application defined property holds. If so, then the messages are delivered; if not, they are discarded, and the sending application is informed. Thus, the application specifies the *certification policy* for optimistic message delivery, and OVS provides the mechanisms for applying the certification and performing the needed compensation should the certification fail.

In particular, OVS generalizes both of the commonly provided Virtual Synchrony semantics: On one side of the spectrum, applications that do not require Sending View Delivery can send optimistic messages that will always be delivered, providing semantics similar to [5, 8, 11]. On the other side of the spectrum, applications that rely on Sending View Delivery may ignore OVS by simply refraining from sending optimistic messages, providing semantics similar to [4, 14, 24]. We have observed, however, that most applications lie somewhere in the middle. Applications that use Sending View Delivery seldom require complete knowledge of the coming view; typical applications are satisfied by weaker constraints. Several examples of applications that benefit from the improved performance of OVS are discussed in Section 3.

We built a version of OVS on top of an existing group communication service, Transis [11]. Transis does not provide Sending View Delivery to applications. However, the communication mechanism used internally by Transis servers does provide Sending View Delivery (at the server level only). 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, the overhead induced by OVS is very small. We describe the implementation and present our performance measurements in Section 4.

The method we used to implement OVS in Transis can be applied in any group communication system that allows multiple processes to send messages concurrently (e.g., Horus [14, 23], Ensemble [15], Relacs [8], and those of [18, 24]). We believe that it is possible to design a similar method for supporting OVS in token-based systems (e.g., Totem [4]), but we have not explored this possibility.

1.1. Evaluating OVS

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 undone (rolled back) infrequently enough that the cost of undoing actions is masked by the gain from optimism.* With OVS, the undoing (or rolling back) of optimistic messages is very small, as these messages are sent when the available bandwidth would otherwise not be utilized, and they are merely dropped.

(Note that messages are not optimistically delivered to the application, hence applications do not need to undo their effects). 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.

The frequency at which optimistic messages are 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 if these changes are not allowed by the application-specified policy. Note that the fraction of optimistic messages that are dropped is highly application dependent. In fact, in Section 3 we show examples of applications that *never* drop optimistic messages.

2. The OVS Programming Model

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

Group communication services that support Sending View Delivery 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 communication 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 follows all 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. The blocking mechanism ensures that every message is delivered in the view in which it was sent (see [14, 18]).

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., [19, 4]) can usually provide an optimistic view that 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 optimistic mode, the application may still receive messages that were sent in the view that it is leaving, but at the same time, the application can optimistically send messages to be conditionally 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; the *viewAck* denotes the end of the optimistic messages for this view.

Messages sent in the normal mode are delivered in the view in which they are sent. Optimistic messages, if delivered, are delivered in the view that immediately follows the view in which they are sent. When the new view is delivered, the group communication service checks whether the optimistic messages should be delivered or not. This is checked by applying an application-provided predicate, *MessageCondition*, to each optimistic message. All the processes evaluate the same *MessageCondition* deterministically. 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 non-delivered optimistic messages via the *discardedMessages* event.

The parameters of the *MessageCondition* predicate include the set of members of the new view, the optimistic view in which the message was sent, and 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., [10, 7], or the transitional view/set of [21, 2, 26, 18]) can provide this information as a parameter to the predicate as well. Section 3 gives some examples of *MessageCondition* predicates.

Note that OVS generalizes both of the commonly provided Virtual Synchrony semantics: it can be used by applications that do not require Sending View Delivery (e.g., [6]) by always evaluating the *MessageCondition* to *true*, and can be ignored by applications that wish to preserve Sending View Delivery semantics in their original form.

3. Examples of Using OVS

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.

3.1. Primary Views

Applications that maintain globally consistent shared state (e.g., [17, 3, 12, 20]) 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 can use Sending View Delivery as follows: messages that update an object are sent only in the 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)
```

Thus, with OVS messages are blocked only when the master site fails, while with Sending View Delivery messages are blocked when *any* site fails or is suspected of failing.

A similar approach can be used for quorum-based algorithms; the predicate can check if the new view contains a quorum of *m.object*'s copies. When an optimistic message is discarded, the sender stores the request until the view changes to a primary one.

3.2. State Transfer

Typical applications of group communication services, (e.g., [2, 25, 16, 17, 3, 20]), sometimes engage in state transfer when 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 Sending View Delivery, or impose it by tagging state transfer messages with views and discarding messages pertaining to old views (see [2]).

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: Sending View Delivery as specified in [21, 12, 16, 18] and provided by group communication services such as [9, 24, 4, 14, 18] provides a much more costly abstraction than is needed for this application. On the other hand, group communication services that do not support Sending View Delivery (e.g., [11, 5]) 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., [17, 3, 12, 20]) refrain from sending messages until all state transfer messages are received. Thus such applications extend the blocking period imposed by the group communication 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 messages in the previous view. Such a set is called the *transitional view/set* in [21, 2, 26, 18]. If the transitional set is a superset of the new view, then such applications need not engage in state transfer (see [2, 26]).

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 discarded, and the application can re-send the information once the state transfer is complete.

3.4. Data Vectors

Sending View Delivery is exploited by applications that send vectors of data corresponding to processes: it allows such applications to send the vector without annotations, by corresponding the ith entry in the vector to the ith member of the current view. This reduces the amount of context information sent with each message and the message processing time (see [14]).

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.

3.5. Causal Multicast

An example of an application that sends vectors of data corresponding to processes is an implementation of causal multicast using *vector clocks* [13]. 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 processes. 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 [13]).

This technique is used for implementing causal group multicast in the ISIS and Horus group communication systems. With Sending View Delivery, 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, then only the vector clock values for the processes in the view need to be included in further messages, and this greatly reduces the overhead.

This implementation of causal multicast was a main influence on the design of the *Weak Virtual Synchrony (WVS)* programming model of Horus [14]. While a view change takes 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). WVS is exploited for causal vectors as follows: in suggested views, processes send vectors that 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: In optimistic mode, processes can send vectors which include entries for all of the members of the optimistic view. As explained in Section 3.4 above, those can be sent without annotations. However, in order to preserve causality, the vector has to include indices corresponding to *all* the view members. Therefore, if the ensuing view is indeed a superset of the optimistic view, the messages can be processed as with WVS. Otherwise, they should be discarded. The message condition is as follows:

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

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

4. Implementation and Performance

We implemented Optimistic Virtual Synchrony on top of the Transis group communication service [11]. We chose Transis because it provides the two semantics we were interested in: it provides the application processes with a service that does not provide Sending View Delivery, and it internally uses a service that does provide Sending View Delivery. We used this service to implement both Sending View Delivery and OVS in Transis, and to provide these semantics to Transis applications. We compared the performance of Transis with support for OVS with the performance of normal Transis as well as with the version of Transis that does support Sending View Delivery.

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 communication service.

2. To compare the performance of OVS with that of a group communication service that provides Sending View Delivery, as well as with the performance of one that does not. By implementing OVS on top 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 OVS 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, Sending View Delivery is supported among the Transis servers.

With OVS, optimistic messages can be sent during this time period. The synchronization messages together with the FIFO ensure 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 1.

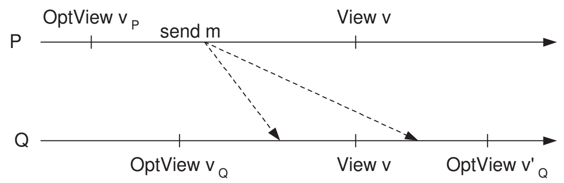

Figure 1. Possible arrival times of optimistic messages at Transis servers.

We added code to Transis in the following places:
1. *When a Transis server initiates a view change.* An *optView* is sent to the application processes. When an application responds with a *flush*, it has entered optimistic mode. The application mode (optimistic or normal) is saved in the OVS process.
2. *When a process sends a message.* 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. *When a process receives a message.* (including loopback 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 Transis.

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 Transis delivers a new view.* Each message in the optimistic message buffer is checked for whether 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 sender 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. Therefore, once the new view is delivered by Transis, the OVS service at the sender immediately retransmits duplicates of these messages to new members that were not in the sender's optimistic view (without waiting for the Transis loss detection mechanism to detect that they are missing).

Since Transis does not support selective sending to individual processes, we re-send the message to the entire view. Note that doing so sends a duplicate of the original message, which processes that had previously received it simply ignore. This is a penalty of using a broadcast mechanism; with point-to-point communication, this could be avoided.

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

Note that OVS can be implemented in a similar manner atop any group communication system in which processes communicate over FIFO channels, can send messages spontaneously without waiting for a token, and send synchronization (or *flush*) messages to each other to denote the end of the message stream in a terminating view. Examples of such systems include [14, 15, 8, 18, 24, 23].

4.2. Performance Measurements

In this section we describe the measured performance of OVS implemented on top of Transis. The tests were run on three 333 Megahertz Sun UltraSparc 5/10s, running SunOS version 5.6. The machines were connected via 10MBit/sec Ethernet. The machines were not being used by others users during these tests. In each test, no fewer than 40,000 messages were sent. All messages in the tests were about 1Kbyte 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 measured two different aspects of OVS: (1) the overhead associated with processing messages; (2) 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.

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 before the message can be delivered to the application process.

We measured the average wall clock time a message spends in each of these stages. As expected, our measurements show that the wire time is the most significant component in the message life-cycle: The average pre-send processing time was consistently around 90 μs for all of the tests we ran. The average wire time was around 1000 μs, and the average pre-delivery processing time on the server side was consistently around 40 μs for all of the tests.

The main performance gain of OVS is the utilization of bandwidth that would be unused with group communication services implementing Sending View Delivery. By utilizing this bandwidth, we mask the pre-send processing time and wire time for optimistic messages, to allow faster delivery of messages after the view change. Our experiments demonstrate that the time gained in faster delivery outweighs any overhead induced by OVS in the pre-delivery processing time[1].

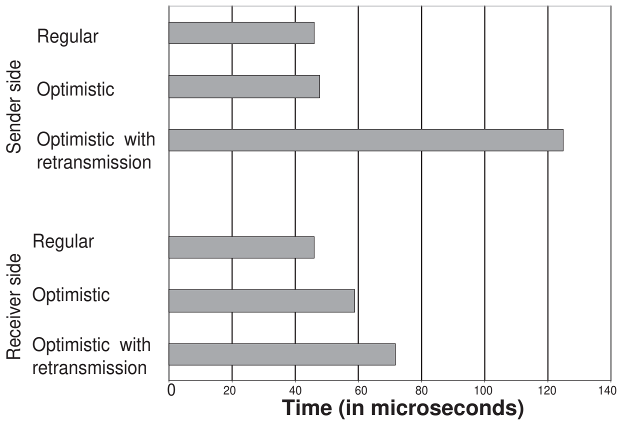

Figure 2. Transis and OVS pre-delivery message processing times.

In Figure 2 we compare the pre-delivery processing time of regular messages in Transis to that of OVS. For OVS, we distinguish between two cases: (1) the new view contains only members that were in the optimistic view, hence no message retransmission is needed; and (2) the new view does contain new members and retransmission is needed[2]. In the second case the pre-delivery time on the sender is larger since it includes the time required to retransmit the message. On the receiver side, extra time is necessary to iterate over the message buffer, evaluate the *MessageCondition*, and change the internal structure of the message. We measured the pre-delivery processing time both at the sender and at the receiver. As expected, with no retransmissions, the processing time was only slightly larger with OVS: around 50 μs at the sender and around 60 at the receiver. With retransmissions, the sender side pre-delivery processing time is almost 125 μs per message, and the receiver side is slightly over 70 μs per message. Retransmissions slow down the sender which has to engage in retransmitting them; they also slow down the receiver since in Transis messages are retransmitted to all of the processes, and therefore the receiver receives duplicate messages.

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

4.2.2 Latency of optimistic messages

The direct benefit of OVS is that it allows messages to be sent during the view change while still providing Sending View Delivery semantics. Transis normally does not provide applications with Sending View Delivery 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 opposed to being buffered at the receiver side with OVS and not being sent at all with a service providing Sending View Delivery. Thus, the latency associated with the communication should be masked by OVS.

To measure this benefit, 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 version of Transis that supports Sending View Delivery, it is the first message sent after blocking; and in the normal Transis, it is the first message buffered during

[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.

[2] This latter case did not naturally occur in our experiments, in which the new view was always identical to the optimistic one. For the sake of measuring it, we had the OVS service retransmit messages although these were not needed.

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 primitives:

1. the normal Transis, without Sending View Delivery;
2. a version of Transis that supports Sending View Delivery;
3. Transis with support for OVS, without the need for retransmission; and
4. Transis with support for OVS, with the need for retransmission.

Figure 3 shows, for one test, the measured $f(p,k)$ for these four primitives. Since p is the sender in this test, 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 primitives other than the version of Transis providing Sending View Delivery cannot be seen separately. We observe that although OVS is as powerful as Sending View Delivery, it allows for communication speeds comparable to a system that does not provide Sending View Delivery.

In our experiments, an average view change took about one second. Running on 10MBit/sec Ethernet, OVS sent approximately 1000 1Kbyte messages during this second. Since with Sending View Delivery this bandwidth is not exploited, there is a build-up of messages following the view change. Figure 3 shows that the build-up lingers on and effects the delivery latency for several seconds. So, messages sent with OVS are still delivered faster than those sent with Sending View Delivery a few seconds after the view change.

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

5. Related Work

The use of optimism in group communication was previously suggested by [22]. There, optimistic assumptions are made about the order in which messages are received in order to quickly provide totally ordered message delivery, whereas in our approach, optimistic assumptions are made about the view. The optimism of [22] is orthogonal to our use, and the approaches could be combined.

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

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. In some of these systems (e.g., Transis [11] and Spread [5]), client membership is implemented as a light-weight layer that communicates with a heavy-weight layer asynchronously using a FIFO buffer. Although the heavy-weight layer supports Sending View Delivery, the asynchrony may cause messages to arrive in later views than the ones in which they were sent. The resulting semantics are too weak for many applications, as illustrated in Section 3 above. In other light-weight group systems (e.g., [23]), the semantics of the heavy-weight layer are preserved, and applications are required to block. OVS can be used to improve the performance of systems of the latter kind.

In order to eliminate the need for blocking while still providing support for a certain type of applications that rely on Sending View Delivery, WVS [14] was introduced. In WVS, every view v is preceded by at least one *suggested view* event. The set of processes in the suggested view is guaranteed to be an ordered superset of v. Sending View Delivery is replaced by the weaker requirement that messages sent in the suggested view be delivered in the next regular view, and processes can continue sending messages while the view change is taking place. 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, applications may still send vectors of data indexed by processes.

One shortcoming of WVS is that it is useful only for applications that are satisfied with knowledge of a superset of the actual view, and does not suffice for 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

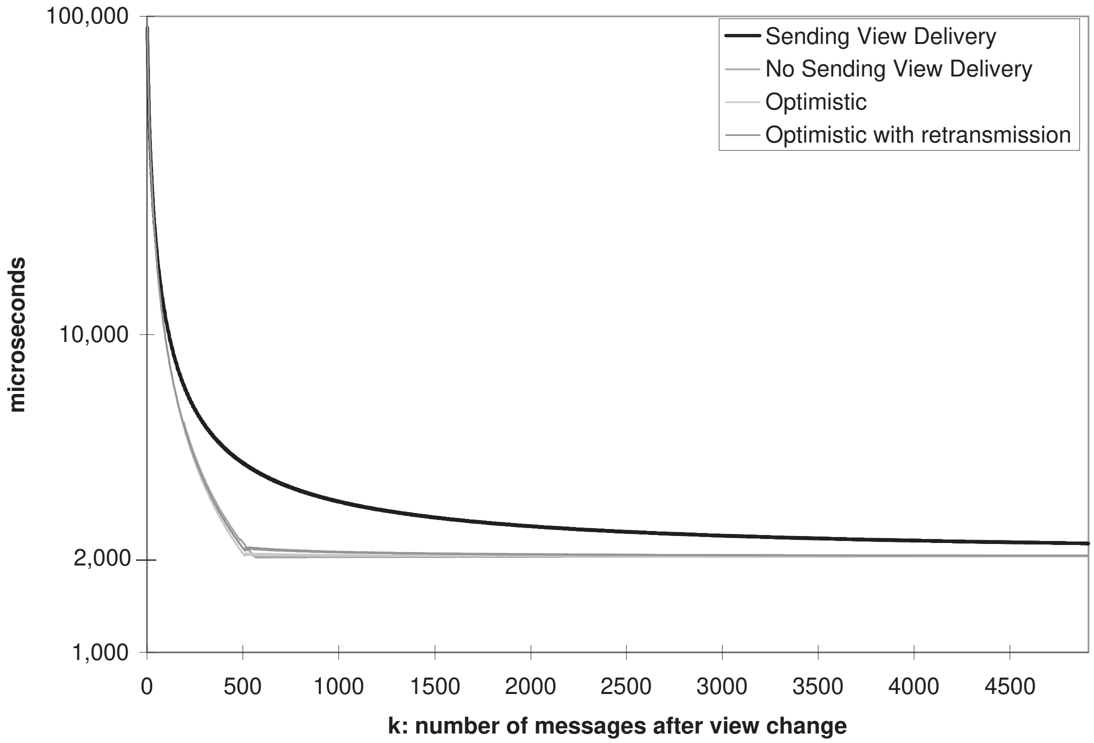

Figure 3. $f(p, k)$ **for sender delivery (in logarithmic scale).**

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 [19]).

6. Conclusions

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

OVS provides applications with the flexibility to determine the certification policy (message condition) as to when optimistic messages should be delivered and when they should be discarded. We have described several different applications that can benefit from OVS. Our examples illustrate how different applications can use OVS with different message conditions. We have observed that applications seldom require that the new view be identical to the optimistic one; typical 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 group communication services that support Sending View Delivery would block. The latency of optimistic messages sent using OVS is significantly smaller than the latency imposed by the blocking period in group communication services that provide Sending View Delivery, and is similar to the latency of messages sent during view changes using a group communication service that does not.

Acknowledgments We thank Roger Khazan and Danny Dolev for many interesting discussions on the material in this paper and the anonymous referees for their insightful comments.

References

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

[2] 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. Systems (ERSADS'97)*, pages 183–192, March 1997.

[3] Y. Amir, D. Dolev, P. M. Melliar-Smith, and L. E. Moser. Robust and Efficient Replication using Group Communication. Tech. Report CS94-20, Inst. of Comp. Sci., Hebrew University, Jerusalem, Israel, 1994.

[4] 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), November 1995.

[5] Y. Amir and J. Stanton. The Spread Wide Area Group Communication System. TR CNDS-98-4, The Center for Networking and Dist. Systems, The Johns Hopkins University, 1998.

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

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

[8] Ö. Babaoğlu, R. Davoli, and A. Montresor. Group Communication in Partitionable Systems: Specification and Algorithms. TR UBLCS98-1, Dept. of Computer Science, University of Bologna, April 1998.

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

[10] F. Cristian and F. Schmuck. Agreeing on Process Group Membership in Asynchronous Distributed Systems. Tech. Report CSE95-428, Dept. of Computer Science and Engineering, University of California, San Diego, 1995.

[11] D. Dolev and D. Malkhi. The Transis approach to high availability cluster communication. *Commun. ACM*, 39(4):64–70, April 1996.

[12] A. Fekete, N. Lynch, and A. Shvartsman. Specifying and using a partitionable group communication service. In *16th ACM Symp. on Prin. of Dist. Computing (PODC)*, pages 53–62, August 1997.

[13] C. Fidge. Logical time in distribution systems. *Computer*, 24(8):28–33, 1991.

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

[15] M. Hayden and R. van Renesse. Optimizing Layered Communication Protocols. Tech. Report TR96-1613, Dept. of Computer Science, Cornell University, Ithaca, NY 14850, USA, November 1996.

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

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

[18] I. Keidar and R. Khazan. A client-server approach to virtually synchronous group multicast: Specifications and algorithms. In *20th Intl. Conf. on Dist. Computing Systems (ICDCS)*, pages 344–355, April 2000.

[19] I. Keidar, J. Sussman, K. Marzullo, and D. Dolev. A Client-Server Oriented Algorithm for Virtually Synchronous Group Membership in WANs. In *20th Intl. Conf. on Dist. Computing Systems (ICDCS)*, pages 356–365, April 2000.

[20] 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 DIStributed Computing (DISC)*, pages 258–272, Andros, Greece, September 1998.

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

[22] F. Pedone and A. Schiper. Optimistic atomic broadcast. In *12th Intl. Symp. on DIStributed Computing (DISC)*, pages 318–332, September 1998.

[23] L. Rodrigues, K. Guo, A. Sargento, R. van Renesse, B. Glade, P. Verissimo, and K. Birman. A dynamic light-weight group service. In *15th IEEE Intl. Symp. on Reliable Dist. Systems (SRDS)*, pages 23–25, Oct. 1996.

[24] A. Schiper and A. Ricciardi. Virtually synchronous communication based on a weak failure suspector. In *23rd IEEE Fault-Tolerant Computing Symp. (FTCS)*, pages 534–543, June 1993.

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

[26] R. Vitenberg, I. Keidar, G. V. Chockler, and D. Dolev. Group Communication Specifications: A Comprehensive Study. Tech. Report CS99-31, Inst. of Comp. Sci., Hebrew University, Jerusalem, Israel, September 1999.