Efficient State Transfer in Partitionable Environments *

Yair Amir† Gregory V. Chockler‡ Danny Dolev§ Roman Vitenberg¶

February 1, 1997

Abstract

Object replication is one of the most useful techniques in distributed computing because it facilitates fault-tolerance and increases the availability of distributed services. Consistent replication is the focus of traditional distributed database applications as well as of Computer Supported Cooperative Work applications. Other applications that can take advantage of consistent object replication are those operating in a mobile environment.

However, in asynchronous environments prone to machines and communication link failures, disconnected replicas may find themselves with different states. The State Transfer problem is to bring such replicas to a consistent state when they re-connect again. Our work presents an efficient State Transfer layer that may serve as a building block in consistent object replication protocols.

The traditional Virtual Synchrony model does not provide sufficient services and guarantees in order to implement the State Transfer layer efficiently. Our solution is based on utilizing the notions of hidden membership and transitional set, which were proposed as basic concepts of the Extended Virtual Synchrony model. In this paper we further elaborate the definitions of hidden membership and transitional set in order to shed light on the subtle aspects of membership service in partitionable environments.

Currently, the protocol is implemented as part of the group membership of Transis communication system.

1 Introduction

Object replication is one of the most useful techniques in distributed computing because it facilitates fault-tolerance and increases the availability of distributed services. However, in asynchronous environments prone to machines and communication link failures, disconnected replicas may diverge with different states. The *State Transfer problem* is to bring such replicas to a consistent state when they re-connect. This paper presents an efficient implementation of the *State Transfer module* that may serve as a building block in object replication protocols. This module was implemented and exploited by applications in the Transis [4] and Totem [6] environments.

Replication is the focus of traditional distributed database applications [19, 5, 1, 2] as well as of *Computer Supported Cooperative Work* [23] applications. The latter includes multimedia and desktop conferencing systems, interactive distributed games and simulations, distributed applications with shared workspace, *etc.* Other applications that employ object replication are those operating in a mobile environment. In such an environment, changes in network connectivity are particularly frequent.

The sooner the state transfer terminates, the sooner the system becomes consistent and returns to its normal operational mode. Many distributed applications neither apply updates nor handle local requests until this step is completed. Therefore, the efficiency of the State Transfer module is important for such applications, and may even be crucial for the time dependent ones. Moreover, for applications operating in a mobile environment, the limited bandwidth and the overhead of broadcasting a message will strongly favor protocols that do not send redundant information.

Basic services that help solving the State Transfer problem are reliable, totally ordered multicast and membership services, typically provided by group communication systems [20, 4, 6, 25, 11, 8, 15, 27]. A straight-forward and widely used protocol that accomplishes state transfer is the one in which every replica

*This work was partially supported by ARPA grant No. OSP#28899-5530 and by the Ministry of Science of Israel, grant number 834-6195

†E-mail: yairamir@cs.jhu.edu, Department of Computer Science, The Johns Hopkins University, Baltimore MD 21218, and the NASA Center of Excellence in Space Data and Information Sciences.

‡E-mail: grishac@cs.huji.ac.il, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem 91904, Israel.

§E-mail: dolev@cs.huji.ac.il, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem 91904, Israel.

¶E-mail: romanv@cs.huji.ac.il, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem 91904, Israel.

multicasts its state upon receiving a membership notification. Each replica gathers all the state messages sent by other members and computes a new common state. This protocol is inefficient because in most cases redundant states are sent. This paper suggests that only a representative of a group of processes with identical states will multicast its state. This representative is chosen without additional communication. The challenge we face is how to identify the set of processes having identical states.

The membership service may be used by a process to determine which processes belong to its group. Unfortunately, a membership service that conforms with the virtual synchrony model is not sufficient. This is because a membership notification received by a replica from such a membership service does not carry enough information to determine which members are guaranteed to have identical states, as demonstrated in Section 4.

Our solution overcomes this difficulty by utilizing the notions of *hidden* membership and *transitional* set, which were proposed as basic concepts of the *Extended Virtual Synchrony model (EVS)* [21, 14]. The transitional membership notification, complementing the regular membership notification, enables each replica to locally determine the set of other members that have the same state. In this paper we further elaborate the definitions of *hidden* membership and *transitional* set in order to shed light on the subtle aspects of membership service in partitionable environments.

Our proposed State Transfer module may be implemented on top of any group communication layer that preserves extended virtual synchrony. Transis [4] and Totem [6] present examples of such systems, where the transitional set is implemented at no additional cost, compared with virtually synchronous group communication.

The *state transfer tool* of Isis [10] was, perhaps, the first to provide a generic solution to this problem, although in a primary partition model. Other work tackling the state transfer problem can be found in [13, 7, 18].

The approach presented in [7] is the closest to ours. This work defines the *shared state problem* in a partitionable environment and contains a detailed survey of its different aspects. It demonstrates that traditional view synchrony cannot provide accurate information regarding which members of the newly installed view have the same state of the shared object. The work introduces an extension of the view synchrony model, called *enriched view synchrony*. However, the main contribution of [7] is a new high level paradigm for building distributed applications, operating in group communication framework. In contrast, the main objective of our paper is the subtleties of the underlying communication model that render the efficient identification of the set of processes with the same state feasible. We show that the extended virtual synchrony model, used by numerous distributed applications [19, 5, 1, 2], allows tackling the State Transfer problem elegantly and efficiently.

2 System Model

2.1 Environment and Failure Model

The system consists of a group of processes, communicating via asynchronous multicast messages. A process may incur a crash failure and restart.

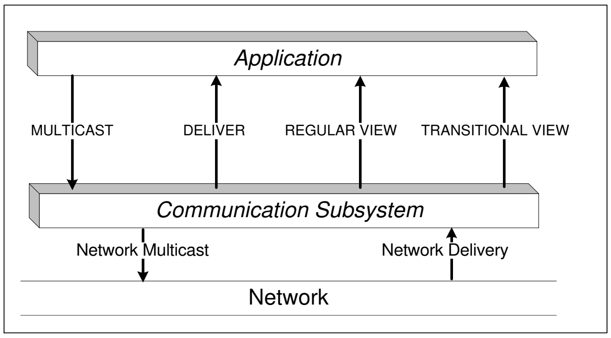

Figure 1: The System Architecture

The network may partition into a finite number of connected components. Two or more components may merge to form a larger connected component. It is assumed that when processors are connected (*i.e.* reside in the same component), each message has a non-zero probability of reaching its destination. It is further assumed that a failure cannot alter the content of a message.

We assume the system model appearing in Figure 1. The State Transfer module is assumed to be part of the application. In the full version of this paper [3] we elaborate on the implementation and on the interface between the application and the module.

2.2 Communication Subsystem

Our model assumes the existence of an underlying communication subsystem that provides a partitionable *membership service* [21, 14, 25, 17, 9] and a *totally ordered multicast service*, and that guarantees certain properties, described in the following sections. All these services and properties are a strict subset of the *Extended Virtual Synchrony* [21] model, supported by the Transis [4] and Totem [6] systems.

2.2.1 Membership Service

Membership is a fundamental paradigm of fault-tolerant distributed computing. Its objective is to maintain the list of currently connected and operational processes and to notify the application each time the list changes. Membership belongs to the same class of problems as distributed Consensus and Atomic Broadcast and, like them, does not have a solution that always terminates [16, 12, 26]. Yet, a membership service is vital for the development of fault tolerant applications and distributed systems that are exposed to the kinds of failures sketched in Section 2.1.

The membership service described below is *partitionable*. This means that several disjoint components of processes may co-exist.

According to the membership service specifications of the Extended Virtual Synchrony model, an agreement value μ consists of a membership set of processes $\bar{\mu}$ and a unique membership identifier that distinguishes between different views with the same set of processes. Once a membership service algorithm, running at one process, decides on some μ, it notifies the upper layer (or the application) by delivering an appropriate message. This notification event is called a *regular view installation* and the value of μ is called a *regular view*. In the rest of the paper we say that some event *occurred in* μ if the last regular view installation, preceding this event, was an installation of μ.

The following properties are part of the delivery of configuration changes requirements of Extended Virtual Synchrony:

Self-inclusion implies that if a process p installs a regular view, then this view includes p as its member.

Termination of membership implies that:

- If a process p installs a regular view, then either every member of that view installs the view, or p installs a new regular view, or p crashes.
- If two processes p and q install the same regular view and p installs the next regular view, then either q installs a new regular view or q crashes.

2.2.2 Totally ordered delivery

The totally ordered multicast service handles MULTICAST and yields DELIVER primitives, as depicted in Figure 1. The communication subsystem accepts MULTICAST requests to send messages, and delivers the messages via DELIVER events in a total order that preserves causality.

2.2.3 Message Delivery

The following are properties of message delivery:

Self-delivery implies that if a process sends a message in some view, this message is delivered to the process in the same view.

Virtual synchrony [24, 11] implies that if two processes install the same two consecutive views V_1 and V_2, then the same set of messages is delivered at the processes between these views. This property is called *Failure Atomicity* in the extended virtual synchrony model [21].

Termination of delivery implies that if two processes p and q install the same regular view and a message is delivered at p in this regular view, then either this message is delivered also at q, or q installs a new regular view, or q crashes.

2.2.4 Extended View

The communication subsystem provides another primitive, called EXTENDED_VIEW (see Figure 1), that extends the regular view by providing more information to the application. Extended view EV consists of a *regular view* $EV.reg$ and a *transitional set* $EV.trans$. The transitional set is an addition to the regular view that, informally speaking, allows a process to know which members of its previous regular view proceed along with it to the next regular view. Essentially, the transitional set is contained in the *transitional view*, which is an innovative paradigm of Extended Virtual Synchrony [21].

In the following discussion, installation of a regular view μ refers to an installation of an extended view EV such that $EV.reg = \mu$.

In order to provide a strict definition, we make use of the notion of *hidden installation*, introduced in [22] and further developed in [14, 3]. Intuitively, to achieve the agreement, the membership protocol running on some process has to give its consent to the next regular view μ. Upon giving the consent, the membership protocol signals this action by an internal event, called *hidden installation* of μ. The content μ of this event is called *hidden view*. A hidden view is not reported to the application. Only when it is known to the membership protocol that other members of $\bar{\mu}$ [1] also gave their consent to μ, it delivers μ as a regular view to the

[1]Recall that $\bar{\mu}$ denotes the set of processes of μ

application. Thus, before the installation of the regular view μ at a process p, all members of $\bar{\mu}$ (including p itself) must have installed μ as a hidden view.

It might happen, however, that after the hidden installation and before hearing the confirmation from the other members of $\bar{\mu}$, the process will detach from them. In this case, the membership protocol will not be able to install μ as a regular view, because it lacks support from the other members. Then, an attempt of achieving the agreement on another regular view μ' will take place, so that μ' will be also installed as a hidden view. This process may be iterated several times, until the agreement is reached. Thus, the last hidden installation before the installation of the regular view μ is always the installation of μ as a hidden view. All hidden installations prior to the hidden installation of μ present an evidence of the failed agreement attempts.

Now we are ready to present the full definition of a transitional set. After having agreed upon the next regular view μ', the communication subsystem running at a process p, computes a transitional set T. This set is included into the next extended view along with μ'. T is computed according to the following rule: Assuming that the previous regular view, installed at p was μ, a process $q \in T$ if and only if

- $q \in \bar{\mu} \cap \bar{\mu'}$ and
- μ was the last regular view delivered at q before μ' was installed at q as a hidden view[2].

2.3 Discussion

Both absolute agreement and accuracy requirements from a partitionable membership service are satisfiable only with a very strong failure detector, which is not realistic in an asynchronous environement [16, 12, 26]. In practice, every virtual synchrony model weakens the agreement requirement. Consequently, a process that decides on a regular view, cannot know whether the decision values of other members of the view are different from this regular view. Nevertheless, a process can learn about these decision values *a posteriori*. The transitional set delivered at a process p along with the current regular view, includes only those members of this view for which their previous regular view is identical to p's previous regular view.

The scenario depicted in Figure 2(a,b), illustrates the meaning of a transitional set. In Figure 2(a) the communication subsystem delivers regular views only, and in Figure 2(b) the processes install extended views. p and q start as one connected component (both install $<1, \{p, q\}>$). Later, p splits from q. q detects this partition first and installs the regular view $<2, \{q\}>$. When the slower process p also detects the fluctuation in the network connectivity and activates the membership protocol, the network reconnects and both processes install $<3, \{p, q\}>$, first as a hidden view, and then as a regular view. Note that it is important to distinguish between $<1, \{p, q\}>$ and $<3, \{p, q\}>$, despite the fact that they have the same membership. Since the hidden installation occured at q in the regular view $<2, \{q\}>$, p and q came to $<3, \{p, q\}>$ from different regular views. Hence, the previous decision value of q is different from that of p and the transitional set delivered along with $<3, \{p, q\}>$ on p, does not include q.

Essentially, the transitional set consists of those members that proceed together through the same consecutive regular views. According to the virtual synchrony property, all of the members of the transitional set deliver the same set of messages between these regular views. Therefore, if some process q belongs to a regular view that is delivered to process p, and to every succeeding transitional set delivered to p hereafter, p knows that it delivered an identical set of messages (and in the same order) as q, between these view changes.

In the following sections we will use this property to guarantee that if p and q have the same state, and subsequently q belongs to a transitional set delivered to p within the next extended view, then p has the same state as q given that both install the same next regular view[3].

As was observed by Cristian [13] and subsequently elaborated by Babaoglu *et al.* [7], the traditional Virtual Synchrony model is not sufficient to derive information regarding which processes proceed together through the same consecutive regular views. The latter work introduces a novel programming paradigm, called *enriched view synchrony* that facilitates the reasoning about the shared state problem using only information that is locally available to processes. While the general methodology proposed in this paper is appealing, the authors do not concern with the subtleties of the model, necessary to implement this concept[4]. Furthermore, the authors do not discuss the issue

[2] By the definition of hidden view, μ' is necessarily installed at q as a hidden view, because it is installed at p as a regular view.

[3] Recall that the installation of a regular view means an installation of an extended view with a *reg* field equal to this regular view.

[4] Actually, [7] relies on the model, defined in [9], in which several properties are unnecessary.

of the efficient implementation of the concept. The Extended Virtual Synchrony model [21], exploited in our paper, is essentially weaker than the model used in [7] and was efficiently implemented in Totem [6] and Transis [26]. At the same time the EVS model is strong enough to solve the shared state problem efficiently. It should be emphasized, that the concept of *enriched view* is easily implementable on the top of our model.

3 The State Transfer Problem

In this section we present the definition of the State Transfer problem. The definition is similar to the one given in [7].

3.1 Object's State

Each instance of the application maintains a replica of the distributed object. Although the definition of an actual object is application dependent, the following structure is generic: An object consists of a state (a set of variables) and a collection of methods to manipulate these variables. Two special methods are $Extract\_State$ and $Merge\_States$. $Extract\_State$ retrieves the object's state. $Merge\_States$ takes several states and merges them into a joint state. This method is an abstraction of the application's semantics of the resulting state once components merge. We assume that this method has the identity property, *i.e.* $Merge\_States(S, S, \ldots S) = S$.

Methods are classified as *update* or *query* methods. The state of an object may be changed only by update methods. All of the methods are triggered by update messages, generated by the application, with the exception of the $Merge\_States$ method that is invoked by the State Transfer module.

The next state of an object is entirely determined by the current state and by the invoked update method. Thus, if two replicas start with an identical state and subsequently apply the same update messages at the same order, their resulting state will be identical.

3.2 State Transfer

A protocol that maintains consistency in a distributed environment may be viewed as having two modes of operation. In the normal mode a protocol receives update messages, delivered by the communication subsystem and executes application dependent actions. When a new regular view is reported by the communication layer, the *State Transfer* module is invoked. This module has the following goal: When previously separated replicas merge, they may have different states of the object. The replicas should be brought to an identical state before they may resume normal operation.

When this task is accomplished, the State Transfer module terminates and the protocol shifts to the normal mode. Note that while in State Transfer, the protocol may face several view changes in a row. When a new regular view is installed, the State Transfer module discards the states collected in the previous view and starts gathering the states in the new view.

Note that in general, the merged state cannot be computed without knowing the previous states of all of the members of the current view.

Formally speaking, the task of state transfer can be formulated as follows: If a process P_i installs a regular view μ, then a state transfer is initiated at P_i so that the following properties hold:

Correctness Suppose that the state transfer terminates at P_i and each member $P_j \in \bar{\mu}$ installs μ. Let us denote the state of the object at P_j when P_j installs μ as S_j. Then, when the protocol terminates, the state of P_i is the result of applying the method $Merge\_States$ on the collection of S_j's for each $P_j \in \bar{\mu}$.

This requirement contains the condition that all of the members of P_i's view install that view. If this condition does not hold, *i.e.* there is some member that does not install the view, the termination of membership property guarantees that P_i installs a new regular view. In this case, the State Transfer module will be invoked again at P_i and will try to achieve correctness in the new view.

Termination If a process P_i does not crash and does not face an additional view installation event for a sufficiently long time, then the protocol terminates.

By termination of state transfer we understand its successful completion. If the current regular view at P_i is hastily followed by the next regular view before state transfer is completed, the State Transfer module will be invoked again and will attempt to achieve termination in the new view.

4 The Solution Highlights

A straight-forward and widely used protocol that accomplishes state transfer, is the one in which each replica multicasts its state upon receiving a membership notification. Each replica gathers all the state messages sent by other members, and computes a new common state. This protocol is inefficient because in most cases redundant states are sent.

Often, the state of a member when it receives a regular view is identical to the state of some subset

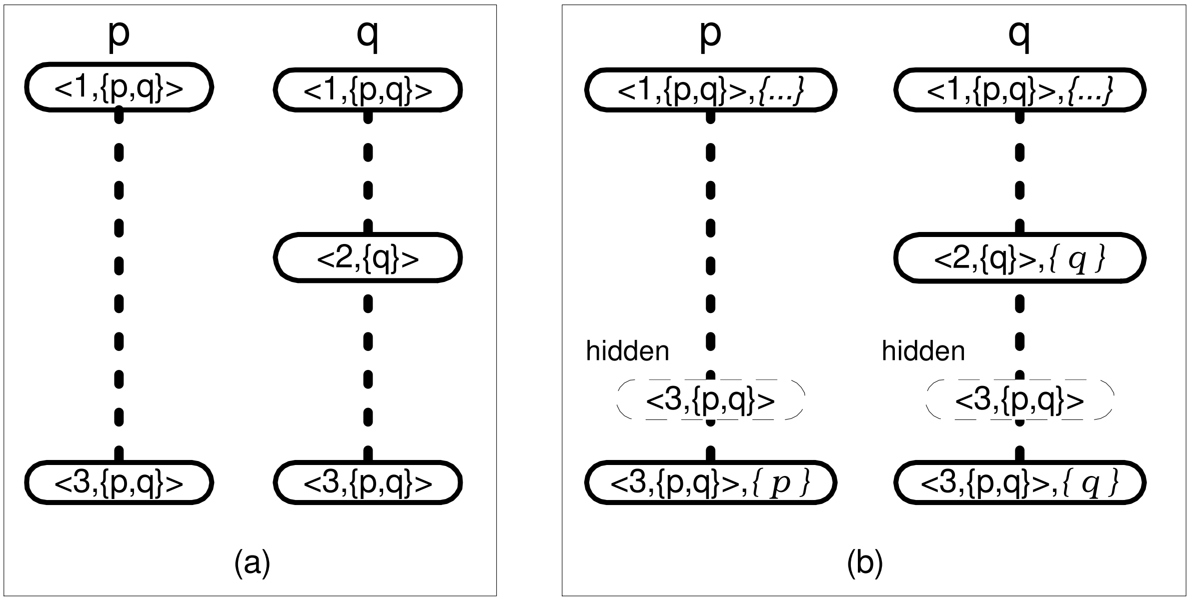

Figure 2: A possible scenario in an asynchronous environment

of the members of this view. It is feasible to split the regular view into equivalence classes by the relation *"having the same state upon entering the view"*. If each member knew the identity of the other members in its class, one representative from each class could be elected to send its state, representing the class. In particular, if there was only one class, no messages need be sent.

Unfortunately, an accurate membership service in an asynchronous environment is impossible [16, 12]. Therefore, there is no way for a process to know the precise membership of its class. However, it is still worthwhile to endeavor different approximations, based on the view messages delivered to the application by the communication subsystem.

4.1 Why Virtual Synchrony Alone Would Not Do

Suppose that the state transfer following the previous membership change has been completed. The most intuitive approach to be adopted by a process p is to use the intersection of the current and previous regular views, reported by the membership service: If the current regular view is contained in the intersection, no messages are sent. Otherwise, this intersection is taken as an approximation of the equivalence class. The representative is chosen from it in a deterministic way.

This approach implicitly relies on the virtual synchrony property. Once the intersection is computed, the process p assumes that its members have passed through the same sequence of view installations as p itself. Thus, according to the virtual synchrony property they have delivered the same set of messages and, therefore, have the same state of the object as p.

This approach works when all the members of the intersection of two regular views install these views, and none of them installs another view in between. However, this is not necessarily true, as explained in Section 2.3. Let us take another look at the scenario, depicted in Figure 2(a). Since p and q do not install the same two consecutive views, the virtual synchrony property is not applicable in this case. Consequently, p and q may have different states of the object when they install $<3, \{p, q\}>$. However, the intersection of two consecutive views installed at p is $\{p, q\}$, so p decides that no state transfer is needed and immediately starts handling updates. Obviously, correctness is violated in this scenario.

4.2 Equivalence Class Approximation By a Transitional Set

In contrast to the intersection of the current and previous views, the transitional set presents a correct approximation. For example, in the scenario above the transitional set delivered along with$< 3, \{p, q\}>$ on p, does not include q (see Figure 2(b)). This way, p can derive that p and q may have different states of the object upon the installation of$< 3, \{p, q\}>$.

According to the virtual synchrony property, all the members of the transitional set deliver the same set of updates between these regular views. Thus, if some process p successfully exchanges states with another process q, and q belongs to every subsequent transitional set delivered to p, then p and q have a consistent state of the object. In the following section this property of the transitional set is referred as *state preservation*.

Note that the information comprised in a transitional set may be obtained by the State Transfer protocol relying solely on the Virtual Synchrony model. However, this requires an additional round of communication at the application level. Such State Transfer protocol would be equally inefficient as the protocol in which each replica multicasts its state. On the other hand, transitional set may be computed by the underlying communication subsystem in the course of reaching the agreement on the next regular view. No extra communication is necessary for this computation.

5 The Protocol

In this section we present the protocol that implements the State Transfer module. The protocol operates on top of the communication subsystem described in Section 2 and conforms with the specifications outlined in Section 3. There is an instance of the protocol running at each process in the system.

5.1 Data Structures

Each instance of the State Transfer protocol maintains the following data structures:

myid – An identifier of this instance of the protocol;

membid – An identifier of the last regular view delivered by the communication subsystem;

S – is the intersection of all the transitional sets delivered since the last successful state transfer. Note, that this is a set of identifiers of the instances for which the object state is the same as the object state of this instance;

StatesV – A vector containing an object state for each member of the current regular view. As the states of the instances arrive, they are accumulated into this vector. When the State Transfer protocol terminates, the states in this array are merged using the *Merge_States* method that yields the combined state of the object;

States_Await – A set of identifiers of the instances for which the object state has not been delivered yet;

Obj – The instance of the replicated object passed to the State Transfer layer by the application.

```
State_Transfer (EV_0, Obj):

Initialization:
    S := EV_0.Trans.members;
    membid := EV_0.Reg.membid;
    if (EV_0.Reg.members = S) return;
    Init_State_Exchange(EV_0.Reg);

Event Loop:
  State Message
  SM: <membid,S,ObjState>
    if (membid = SM.membid) then
      for each i in S do
          StatesV[i] := ObjState;
      od
      States_Await := States_Await \ SM.S;
      if (States_Await = ϕ ) then
          Obj.Merge_States(StatesV);
          return;
      fi
    fi

  Extended View
  EV: <Reg, Trans>
    membid := EV.Reg.membid;
    S := S ∩ EV.Trans.members;
    if (EV.Reg.members = S) return;
    Init_State_Exchange(EV.Reg);
```

Figure 3: The State Transfer Procedure

5.2 The State Transfer Procedure

The State Transfer procedure depicted in Figure 3 is called by the application when a new extended view is delivered by the communication subsystem. The following two parameters are passed to the procedure (see Figure 3): a new extended view EV_0 and the object *Obj*.

The procedure initializes the set S with the transitional view's members, and compares the regular view membership with the membership of the set S. If S consists of the same members as the regular view (i.e. no new processes have joined), there is no need to exchange states. Otherwise, the auxiliary INIT_STATE_EXCHANGE procedure, shown in Figure 4, is invoked. This procedure initializes the *States_Await*

```
procedure INIT_STATE_EXCHANGE(Reg):

  States_Await := Reg.members;
  Allocate StatesV according to the size of Reg.members;
  if (myid = min_{id∈S}(id)) then
      ObjState := Obj.EXTRACT_STATE();
      MULTICAST state message <membid,S,ObjState>
          to all the Object's replicas;
  fi
```

Figure 4: The State Transfer Initializing Procedure

set with the members of the regular view and allocates a space for the *StatesV* array, sufficient to hold all the states of the regular view members. Then, a member of S with the smallest identifier is chosen as a representative of all the instances in S. According to the state preservation property, all the members of S have the same state of the object. Thus, only the representative multicasts the object state on behalf of all members of S.

Eventually, the procedure enters the event loop where the following two event types are handled: delivery of a *state* message multicast by an instance of the protocol and delivery of *extended* view by the underlying communication subsystem.

The procedure remains in the event loop until the received state messages represent all the members of the new regular view. When this condition is met, the accumulated states are merged using the *Merge_States* method. The state transfer is then completed, and the control returns to the application.

If the communication subsystem delivers new extended views before all the states are gathered, the protocol updates its S set to include **only** those members that have survived since the last successful state transfer (and therefore, have the consistent state), discards the states collected in the previous view and retries gathering the states in the new view.

Acknowledgments

We are thankful to David Breitgand, Idit Keidar and anonymous referees for their helpful comments and suggestions.

References

[1] 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 (LNCS 774)*, June 1993.

[2] Y. Amir, D. Breitgand, G. Chockler, and D. Dolev. Group Communication as an Infrastructure for Distributed System Management. In *International Workshop on Services in Distributed and Networked Environment*, number 3rd, June 1996. To appear.

[3] Y. Amir, G. Chockler, D. Dolev, and R. Vitenberg. Efficient State Transfer in Partitionable Environment. TR 97-1, Institute of Computer Science, The Hebrew University of Jerusalem, February 1997.

[4] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A Communication Sub-System for High Availability. In *22nd Annual International Symposium on Fault-Tolerant Computing*, pages 76–84, July 1992.

[5] Y. Amir, D. Dolev, P. M. Melliar-Smith, and L. E. Moser. Robust and Efficient Replication Using Group Communication. TR CS94-20, Institute of computer science, The Hebrew University of Jerusalem, 1994.

[6] 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. on Comp. Syst.*, 13(4), November 1995.

[7] O. Babaoglu, A. Bartoli, and G. Dini. On Programming with View Synchrony. In *Intl. Conference on Distributed Computing Systems*, number 16th, pages 3–10, May 1996. Also available as technical report UBLCS95-15, Department of Computer Science, University of Bologna, 1995.

[8] O. Babaoglu, R. Davoli, L. Giachini, and M. Baker. RELACS: A Communication Infrastructure for Constructing Reliable Applications in Large-Scale Distributed Systems. TR UBLCS94-15, Laboratory of Computer Science, University of Bologna, 1994.

[9] O. Babaoglu, R. Davoli, and A. Montresor. Failure detectors, group membership and view-synchronous communication in partitionable asynchronous systems. TR UBLCS95-18, Department of Conmputer Science, University of Bologna, November 1995.

[10] K. P. Birman, R. Cooper, T. A. Joseph, K. Marzullo, M. Makpangou, K. Kane, F. Schmuck, and M. Wood. *The ISIS System Manual*. Dept of Computer Science, Cornell University, Sep 90.

[11] K. P. Birman and T. Joseph. Exploiting Virtual Synchrony in Distributed Systems. In *11th Ann. Symp. Operating Systems Principles*, pages 123–138, Nov 87.

[12] T. D. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost. On the Impossibility of Group Membership. In *Annual ACM Symp. on Principles of Distributed Computing*, pages 322–330, May 1996.

[13] Flaviu Cristian. Group, majority, and strict agreement in timed asynchronous distributed systems. In *International Symposium on Fault-Tolerant Computing*, number 26th, 1996.

[14] D. Dolev, D. Malki, and H. R. Strong. A Framework for Partitionable Membership Service. In *Annual ACM Symp. on Principles of Distributed Computing*, May 1996. Full version vailable as TR94-6, Inst. of Comp. Sci., the Hebrew University of Jerusalem.

[15] P. D. Ezhilchelvan, A. Macedo, and S. K. Shrivastava. Newtop: a fault tolerant group communication protocol. In *International Conference on Distributed Computing Systems*, number 15th, June 1995.

[16] M. Fischer, N. Lynch, and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32:374–382, April 1985.

[17] Roy Friedman and Robbert Van Renesse. Strong and Weak Virtual Synchrony in Horus. TR 95-1537, dept. of Computer Science, Cornell University, August 1995.

[18] J. H. Howard and S. Katz. Reconciliations. In *Annual ACM Symp. on Principles of Distributed Computing*, number 13th, pages 14–21. ACM, August 1994.

[19] I. Keidar and D. Dolev. Efficient Message Ordering in Dynamic Networks. In *Annual ACM Symp. on Principles of Distributed Computing*, pages 68–76, May 1996.

[20] C. P. Malloth, P. Felber, A. Schiper, and U. Wilhelm. Phoenix: A Toolkit for Building Fault-Tolerant, Distributed Applications in Large Scale. In *Workshop on Parallel and Distributed Platforms in Industrial Products*, October 1995.

[21] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended Virtual Synchrony. In *Intl. Conference on Distributed Computing Systems*, number 14, pages 56–65, June 1994. Also available as technical report ECE93-22, Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA.

[22] A. M. Ricciardi and K. P. Birman. Using Process Groups to Implement Failure Detection in Asynchronous Environments. In *proc. annual ACM Symposium on Principles of Distributed Computing*, pages 341–352, August 1991.

[23] Tom Rodden. A survey of CSCW systems. *Interacting with Computers*, 3(3):319–353, 1991.

[24] A. Schiper and A. Sandoz. Uniform Reliable Multicast in a Virtually Synchronous Environment. In *IEEE Proc. of the 13th Intl. Conf. on Distributed Computing Systems*, pages 561–568, May 1993.

[25] R. van Renesse, T. M. Hickey, and K. P. Birman. Design and Performance of Horus: A Lightweight Group Communications System. TR 94-1442, dept. of Computer Science, Cornell University, August 1994.

[26] R. Vitenberg, G. Chockler, D. Dolev, and I. Keidar. Partitionable Membership: Bridging the Gap Between the Theory and Practice. Tr, Institute of Computer Science, The Hebrew University of Jerusalem, February 1997. To appear.

[27] B. Whetten, T. Montgomery, and S. Kaplan. A high perfomance totally ordered multicast protocol. In K. P. Birman, F. Mattern, and A. Schipper, editors, *Theory and Practice in Distributed Systems: International Workshop*, pages 33–57. Springer, 1995. Lecture Notes in Computer Science 938.