System Support for Partition-Aware Network Applications

Özalp Babaoğlu   Renzo Davoli   Alberto Montresor   Roberto Segala

Department of Computer Science
University of Bologna
40127 Bologna, Italy

Abstract

Network applications and services need to be environment-aware *in order to meet non-functional requirements in increasingly dynamic contexts. In this paper we consider* partition awareness *as an instance of environment awareness in network applications that need to be reliable and self-managing. Partition-aware applications dynamically reconfigure themselves and adjust the quality of their services in response to partitioning and merging of networks. As such, they can automatically adapt to changes in the environment so as to remain available in multiple partitions without blocking, albeit with reduced or degraded functionality. We propose a system layer consisting of group membership and reliable multicast services that provides systematic support for partition-aware application development. We illustrate the effectiveness of the proposed interface by solving several problems that represent different classes of realistic network applications.*

1. Introduction

Functional requirements, which define how output values are related to input values, are usually sufficient for specifying traditional applications. For modern network applications, however, non-functional requirements can be just as important as their functional counterparts: the services that these applications provide must not only be correct with respect to the functional requirements, they must also be delivered with acceptable "quality" levels. Reliability, timeliness and configurability are examples of non-functional requirements that are of particular interest to network applications.

A correct application satisfies its functional requirements in all possible operating environments: it just may take more or less time to do so depending on the characteristics of the environment. On the other hand, there may be operating environments in which it is impossible to achieve non-functional properties beyond certain levels. For this reason, non-functional requirements of network applications define quality *intervals* rather than absolute values that are considered acceptable. In order to deliver quality levels that are both feasible and acceptable, network applications need to be aware of the environment in which they are operating and use this information to dynamically modify their behavior. We call this capability of an application *environment awareness*.

By their nature, network applications for mobile computing, data sharing or collaborative work involve cooperation among multiple sites. For these applications, which are characterized by reliability and configurability requirements, partitionings of the communication network into several partitions is an extremely important aspect of the environment. The nature of a partitioning will determine the quality for the application in terms of which services are available where, and at what performance levels. In other words, partitionings may result in service *reduction* or service *degradation* but need not necessarily render application services completely unavailable. Informally, we define the class of *partition-aware* applications as those that are able to make progress in multiple concurrent partitions without blocking.

Service reduction and degradation that are unavoidable during partitionings depend heavily on the application semantics and establishing them is beyond the scope of this paper. For certain application classes with strong consistency requirements, it may be the case that all services have to be suspended completely in all but one partition. This situation corresponds to the so-called *primary-partition* model [16, 12] that has traditionally characterized partitioned operation of network applications. In this paper we focus on *system services* that support partition awareness such that continued operation of network applications is not restricted to a single partition but may span multiple concurrent partitions. The system provides the necessary abstractions such that the application itself can decide which of its services will be available in each partition and at what quality levels.

Our methodology is based on the *process group* paradigm [12, 7] suitably extended to partitionable systems. Members of a group cooperate in order to implement a given network application. The methodology unifies all relevant aspects of environment awareness (process crashes/recoveries, network partitionings/merges) in a single abstraction: group membership. Partition-aware applications are programmed so as to reconfigure themselves and adjust their behavior using the current membership of the group as input. The perception that each process has of the group's membership needs to be constructed with care, since otherwise, inconsistencies may compromise functional requirements or result in quality levels that are lower than what is feasible. In Section 4.1 we specify a *partitionable group membership service* (PGMS) that forms the basis of our support layer. As illustrated in Section 5.1, PGMS alone may be sufficient for programming a small class of self-configuring network services. To support a broader class of applications that require closer cooperation, in Section 4.2 we augment PGMS with a reliable multicast communication service satisfying *view synchrony* semantics.

The contributions of this paper are to argue that partition awareness is an important attribute of future network applications and to show that the view synchrony service we propose is indeed useful for supporting it. We do so by developing partition-aware solutions to several abstract problems. Each problem is representative of a particular class of realistic applications. The advocated methodology not only results in simple solutions to non-trivial applications, it also admits simple proofs of their correctness.

2. Related Work

Numerous systems have been proposed or built recently as infrastructures for supporting applications in partitionable environments. Most of these systems are based on the process group paradigm and include Horus [17], Transis [10], Totem [15], Relacs [5], and NavTech [9] as some of the more notable examples. There have also been several attempts at developing realistic applications based on the services provided by these systems: Transis used to implement services that guarantee global ordering of messages assuming that a majority of the processes are connected and alive [1, 13]; Transis used to implement simple system management services [2]; Horus used to implement a replicated state machine that replies to all requests in the majority partition but replies only to a restricted class of requests in minority partitions [11]; NavTech used to manage fault tolerance in groupware applications [8]. In most cases, these applications are based on the primary-partition model that is created on top of a partitionable service. As such, progress in these applications is limited to a single partition. The only exception to this observation is the groupware application described in [8]. The work reported in this paper differs from the above proposals in several important ways. First, we consider systematic support for partition awareness in a wide range of application areas rather than one specific area. Second, the applications we consider take full advantage of partition awareness and remain available in all partitions rather than only a majority-based primary partition.

3. System Model

The system comprises a set of processes that communicate by exchanging messages through a network. The system is asynchronous in the sense that neither communication delays nor relative process speeds can be bounded. Practical distributed systems often have to be considered as being asynchronous since transient failures, unknown scheduling strategies and variable loads on the computing and communication resources make it impossible to bound delays.

In the absence of failures, the network is logically connected and each process can communicate with every other process. Processes may fail by *crashing* whereby they halt prematurely. Crashed processes may *recover* and resume their execution after repairs. Communication failures may provoke a *partitioning* of the network whereby processes are split across two or more *partitions*.[1] A partitioning is described by a set of partitions and characterizes the global communication state of the system. Each partition, on the other hand, is described by a set of processes with the property that processes within the same partition are able to communicate while those across different partitions cannot communicate. Upon repairs, multiple partitions may *merge* to form a single larger partition.

Intuitively, partitions correspond to maximally connected components of the logical graph representing the "reachable" relation among processes. As such, they can be defined only in the context of specific communication primitives. For example, two processes may appear to belong to two different partitions with respect to "ping" messages, but the same two processes may appear in the same partition when communicating through email. This is because the two communication services being considered have significantly different message buffering, timeout and retransmission properties. For pings, even a short, transient communication failure may be perceived as a partitioning but communication failures have to persist for an extended period (typically 3 days) before provoking a partitioning with respect to email.

[1] In addition to communication failures, other situations including incorrect or inconsistent routing tables may lead to partitionings. And if the network is not fully connected, process crashes may also result in partitionings.

4. Support for Partition-Aware Applications

Our methodology is based on the *process group* paradigm with suitable extensions to systems that admit partitioning. In this methodology, processes initiate their collaboration towards a given application by *joining* as members a named group. Later on, a process may decide to terminate its collaboration by explicitly *leaving* the group. In the absence of failures, the *membership* of a group comprises those processes that have joined but have not left the group. In addition to explicit joins and leaves, a group's membership will vary also due to failures and repairs. A *partitionable group membership service* (PGMS), specified in the next section, tracks all changes in the group's membership and installs them as *views* at processes through $vchg()$ upcalls. Installed views are abstractions of the environment with respect to process crashes/recoveries and network partitionings/merges. Ideally, there should be a common view of the group's membership that is shared by all of its members and this view should include exactly those members that are currently operational. This is clearly not feasible in a partitionable system where processes in different partitions will have different perceptions of the membership for a given group. Nevertheless, the PGMS we specify is useful and guarantees that each installed view is shared by all of its components and corresponds to an actual partition of a partitioning.

Group members communicate through reliable multicasts by invoking the primitive $mcast()$. Multicast messages are delivered to processes through $dlvr()$ upcalls. The relationship between multicast message deliveries and the sequence of installed views is formalized as *view synchrony* in Section 4.2. Views and multicast messages are labeled in order to be globally unique. Given a view v, we write $\overline{v}$ to denote its composition as a set of process names. The *current view* of process p at time t is the last view to have been installed at p before time t. Events are said to occur *in the view* that is current. View w is called the *immediate successor of v at p* if p installs w in view v. View w is called an *immediate successor* of v if w happens to be the immediate successor of v at some process p. The transitive closure of the immediate successor relation is called the *successor* relation. Two views that are unrelated through successor relation are called *concurrent*.

4.1. Partitionable Group Membership Service

The first layer of our support architecture governs view installations and view compositions related to group membership. Installed views represent the perception of the group's membership that is shared by all processes in the view's composition. In other words, there is agreement among processes on the composition of the view that they will install. Furthermore, processes that install a given view are exactly those within a given partition. We have formalized these ideas leading to a specification for partitionable group membership in asynchronous systems [6]. For sake of brevity, we state our specification informally, omitting the discussions that motivate it. For the same reason, the specification considers group membership changes due to failures and repairs, omitting those due to joins and leaves.

GM1 (View Agreement) *(i) If process p installs view v and its immediate successor w, both containing q, then p installs view w only after q has installed v. (ii) Suppose correct process p installs view v containing some process q. If the current view of q after some time becomes permanently distinct from v, then p will eventually install a new view as an immediate successor to v.*

GM2 (View Accuracy) *If there is a time after which process q remains reachable from some correct process p, then eventually the current view of p will always include q.*

GM3 (View Completeness) *If there is a time after which all processes in some partition Θ cannot communicate with the rest of the group, then eventually the current view of every correct process not in Θ will never include any process in Θ.*

GM4 (View Integrity) *Every view installed by a process includes the process itself.*

GM5 (View Order) *The order in which processes install views is such that the successor relation is a partial order.*

Properties GM1–GM5 together define a *partitionable group membership service* in asynchronous systems.

4.2. View Synchrony Service

The class of partition-aware applications that can be programmed using PGMS alone can be characterized as configuration management. In general, network applications require closer cooperation that is facilitated through communication among group members. In this Section extend the group membership service of the previous Section with a reliable multicast primitive. The resulting service is called *view synchrony* and integrates delivery of multicast messages with the installation of views. Informally, all processes that survive together from one view to another deliver the same set of messages. In other words, there is agreement among the processes on the set of messages that they should deliver in a view before they can install a common next view.

VS1 (Message Agreement) *Given two views v and w such that w is an immediate successor of v, all processes belonging to both views deliver the same set of multicast messages in view v.*

VS2 (Uniqueness) *Each multicast message, if delivered at all, is delivered in exactly one view.*

VS3 (Merging Rule) *Two views that merge to a single view must have disjoint compositions.*

VS4 (Message Integrity) *Each process delivers a message at most once and only if some process actually multicast it earlier.*

VS5 (Local FIFO Order) *If a process p multicasts message m before it multicasts message m', then p does not deliver m' unless it has delivered m.*

VS6 (Liveness) *(i) A correct process always delivers its own multicast messages. (ii) Let p be a correct process that delivers message m in view v that includes some other process q. If q never delivers m, then p will eventually install view w that excludes q as immediate successor to v.*

VS7 (Immediate Local Delivery) *Messages multicast by a correct process p while handling the upcall $vchg(v)$ are delivered by p in view v.*

Properties GM1–GM5 together with VS1–VS7 define a *view synchrony service* in asynchronous systems.

5. Programming Partition-Aware Applications

In this section, we first illustrate how group membership alone can be sufficient to program simple network applications that require no communication. We then consider more realistic applications that require cooperation through communication and show how view synchrony can be used to program them. The problems are stated abstractly in order to hide irrelevant details, and can be instantiated as numerous realistic partition-aware applications. The solutions we present deliberately avoid optimizations that are often possible but would only complicate their presentations.

5.1. Partitionable Service Activator Application

We describe a simple network application for which PGMS alone is sufficient as the support layer. Consider a network service for diffusing a continuous stream of data (e.g., audio, video, stock quotes, news headlines) to a collection of subscribers. The diffusion can be provided by any one of a set of *servers* that have access to the data source. The service should be available in every partition that contains at least one server; furthermore, to minimize wasted resources, multiple active servers within the same partition should be avoided. New servers may be added and existing ones removed at will by an administrator. The goal is to devise a *service activator* algorithm such that a server can decide when it should be active and when it should be passive.

```
procedure PartitionableServiceActivator()
  server_at ← p
  ActivateService()

  while true do
    wait-for vchg(v)
    if server_at = p and Min(v̄) ≠ p then
      DeactivateService()
    if server_at ≠ p and Min(v̄) = p then
      ActivateService()
    server_at ← Min(v̄)
  od
```

Figure 1. Partitionable Service Activator.

A solution must activate a new server if the current one is removed from the system, if it crashes or if it ends in another partition. And when a new server is added, a crashed server recovers or when partitions merge, redundant instances of active servers should be deactivated. During transition periods, it is possible that some partitions contain zero or more than one active server. However, such periods should have bounded duration.

Figure 1 illustrates a partition-aware solution to the service activator problem based on PGMS. The collection of servers form a group. In this way, all relevant events for service activation/deactivation are transformed into view changes. Thus, the algorithm reduces to the management of $vchg()$ upcalls for the group. We assume there is a total ordering among server names such that the function $Min(S)$ returns the smallest name among the set S. At server p, local variable $server\_at$ identifies the server that p believes to be active. A server starts out in the active state and blocks waiting for view change upcalls. This choice for the initial state is arbitrary since it will last only until the first view change reporting the current membership of the group. To handle a view change, the smallest server in the view composition is chosen as the new active server. If p results as the new active server while it had been passive in the previous view, it is activated through $ActivateService()$. If, on the other hand, p had been active in the previous view but some other server is designated to be active in the new one, then p is made passive through $DeactivateService()$. In all other cases, the state of p remains unchanged with respect to being active or passive.

Given the simplicity of the algorithm and the properties of PGMS, it is easy to argue that under stable conditions, each partition containing at least one server will eventually have exactly one active server.

5.2. Partitionable Chat

Consider a service, not unlike Internet Relay Chat (IRC), for holding a discussion among a collection of users. Users

```
procedure PartitionableChat()
  view_comp ← {p}   % Current view composition
  threadset ← ∅      % Set of open threads
  nthread ← 0        % Number of open threads

  while true do
    wait-for event
    case event of

      vchg(v):
        if (v̄ ⊈ view_comp) then
          foreach id ∈ threadset do
            mcast(⟨UPDATE, p, id, thread[id].vc⟩)
        output("Old participants:", v̄ ∩ view_comp)
        output("New participants:", v̄)
        view_comp ← v̄
        foreach id ∈ threadset do
          thread[id].msg ← ∅

      dlvr(⟨UPDATE, q, id, ts⟩):
        CheckNewId(id)
        foreach (q ∈ view_comp − {p}) do
          if ts[q] > thread[id].vc[q] then
            thread[id].vc[q] ← ts[q]
        CausalDelivery(id)

      newthread(m):
        nthread ← nthread + 1
        id ← CreateNewId(p, nthread)
        CheckNewId(id)
        thread[id].lsent ← 1
        ts ← thread[id].vc
        ts[p] ← thread[id].lsent
        mcast(⟨MESSAGE, id, m, ts, p⟩)

      shout(id, m):
        if id ∈ threadset then
          thread[id].lsent ← thread[id].lsent + 1
          ts ← thread[id].vc
          ts[p] ← thread[id].lsent
          mcast(⟨MESSAGE, id, m, ts, p⟩)
        fi

      dlvr(⟨MESSAGE, id, m, ts, q⟩):
        CheckNewId(id)
        thread[id].msg ← thread[id].msg ∪ {(m, ts, q)}
        CausalDelivery(id)
    esac
  od

  function Deliverable(vc, ts, q)
    return (vc[q] = ts[q] − 1) and
           (∀r ∈ view_comp, r ≠ q : vc[r] ≥ ts[r])
```

Figure 2. Partitionable Chat (Part a).

```
  procedure CheckNewId(id)
    if id ∉ threadset then
      threadset ← threadset ∪ {id}
      thread[id].vc ← (0, . . . , 0)
      thread[id].lsent ← 0
      thread[id].msg ← ∅
    fi

  procedure CausalDelivery(id)
    foreach (m, ts, q) ∈ thread[id].msg do
      if Deliverable(thread[id].vc, ts, q) then
        output(id, m)
        thread[id].msg ← thread[id].msg − {(m, ts, q)}
        thread[id].vc[q] ← ts[q]
      fi
```

Figure 3. Partitionable Chat (Part b).

may contribute to the discussion by *creating* a new thread or by *shouting* messages in an existing thread.[2] Messages are potentially addressed to every user who has joined the discussion. Upon a partitioning, the discussion may continue among users in each of the partitions. Users are informed about others with whom they are currently chatting. In some sense, this application extends the notion of partition awareness all the way up to the user level. Shouted messages have to satisfy agreement, integrity, uniqueness and liveness properties of view synchrony messages. Furthermore, messages shouted within the same partition and belonging to the same discussion thread should be seen in an order that is consistent with causal precedence. No requirements are placed on message threads that span multiple partitions. In other words, upon merging, a user may miss some messages that were shouted in other partitions. For this application, we consider it unreasonable to require causal order at a global level since this would force a user returning to the discussion after having been isolated to listen to the entire discussion that occurred during his absence before being able to resume.

Figures 2-3 contain the code of our partitionable chat algorithm. A user invokes the primitive $newthread(m)$ to start a new thread of discussion whose first message is m, and the primitive $shout(id, m)$ to shout message m relative to the thread identified by id. Users receive messages shouted by others and control information from the system through $output()$ events. In order to distinguish between messages related to distinct threads, all multicast messages are tagged with a thread identifier; moreover, each user maintains a different set of variables for each known thread. Causal delivery is guaranteed through vector clocks [14].

[2] For sake of brevity, we do not consider the possibility of closing open threads. The algorithm we present may be extended by allowing users to perform *close-thread* events and by establishing a policy for resolving conflicts due to threads closed in one partition while remaining open in others.

User u increments the local component of its vector clock whenever u delivers one of its own messages. Each multicast message is timestamped with the current vector clock of the sender, except that the entry corresponding to the sender is replaced by the total number of messages the sender has multicast. This is motivated by the fact that the sender of a message will itself output the message not immediately after the multicast, but only after having delivered it (otherwise messages could be output in different views, due to the fact that messages are not guaranteed to be delivered in the view in which they have been multicast).

When a message is delivered, it is stored in a buffer until it can be output so as to maintain causal precedence within the discussion. Upon a view change, the user is informed about others that are currently in the discussion and those with which the communication is impossible. Two types of view changes need to be considered: if the new view is a contraction of the previous one, no further action is needed; users that survived from the previous view to the new one have delivered and output the same set of messages, and thus maintain the same vector clock values for each other user in the new view. If the view change represents an expansion, an additional merging protocol is needed: two merging partitions may have different sets of active threads, or may have output different messages for the same thread. For these reasons, upon a view expansion, users multicast for each known thread a message containing the thread identifier and the corresponding value of the vector clock. When a user delivers such a message, the thread is added to the local list of threads (if unknown) and the corresponding vector clock is updated. In this manner, users will know exactly the same set of threads, and will be able to output all messages delivered during the new view without incurring in causal inconsistencies due to some past partitioning.

We outline how our solution exploits view synchrony. Properties GM1, GM3 and VS6 guarantee that each user will be informed of other users with whom discussion is impossible. And Property GM2 ensures that this will happen only for those users who are effectively partitioned. Properties VS1 and VS7 are used in the vector clock merging protocol. Users surviving from one view to the same next view obtain identical vector clocks after the installation of the new view. Without this guarantee, it would be complex to reconstruct the new vector clocks necessary for users to deliver new messages. Furthermore, Property VS1 guarantees that surviving users will deliver (and consequently output) the same set of messages during the first view. By Property VS4, all output messages have been previously multicast. Property VS3, combined with Property GM5, is used to avoid scenarios in which different overlapping views merge to a single common view. If such scenarios were admitted, two users p and q could output different sets of messages even though user q always installed views including p.

5.3. Partitionable Parallel Computation

Consider a time-intensive computation such as ray tracing, prime factorization or weather forecasting. The computation can be decomposed into a number of subcomputations that can be carried out independently by a collection of *workers*. New workers may be added and existing ones removed at will. The computation and all relevant input data are known ahead of time to all possible workers. The goal of the *parallel computation* problem is to conclude the computation in as short a time as possible despite crashes, recoveries, partitionings and merges.

Our solution to the problem is illustrated in Figures 4 and 5. Workers that will perform work on behalf of the computation form a group. Within each view, the total work is equally distributed among the workers. During normal operation, each worker carries out the subcomputations assigned to it and diffuses the results through multicast messages. There are two types of view changes that need to be considered. If the new view represents a contraction of the previous one due to partitionings or process crashes, and the workers in the previous view know the same set of results, view synchrony guarantees that at the beginning of the new view all workers know the same set of results as well. Thus, it is sufficient to perform a redistribution of the uncompleted work. If, on the other hand, the new view represents an expansion due to network merges or process recoveries, then an information merging protocol is needed so as not to repeat subcomputations that may have already been completed by other workers. The merging protocol may also be necessary in case the workers disagree on the set of completed results due to an incomplete execution of a previous merging protocol. Upon a view expansion, each worker p redistributes the work among other workers that know the same set of results as itself (note that the work cannot be distributed among *all* workers of the new membership since they may know different set of results; thus, they could distribute inconsistently the uncompleted work). Then, each worker p elects a leader c among the set of all workers that know a superset of the results known to p itself. Leader c acts as a representative for all workers that know a subset of the results known to c, and it multicasts a message containing the results that could be unknown to other workers in the view. The choice of electing a leader is motived by efficiency reasons: in this way, only one worker for each merging partition performs the multicast. When the merging protocol is concluded (each worker has delivered one merging message for each of the partitions that have merged in the new view), a new redistribution of the uncompleted work is performed among all workers in the

```
procedure PartitionableParallelComputation(total)
  my_work ← total       % Subcomputations assigned to p
  view_comp ← {p}       % Current view composition
  lset ← {p}            % Proc. that know less results
  mset ← {p}            % Proc. that know more results
  res ← (∅, ..., ∅)     % Array of known results
  new_res ← ∅           % Results delivered in this view
  dist_res ← ∅          % Results merged in this view

  cobegin

    || Task 1:
    while true do

      wait-for event
      case event of

        vchg(v̄):
          foreach q ∈ view_comp ∩ v̄ do
            res[q] ← res[q] ∪ (new_res ∪ dist_res)
          view_comp ← v̄
          new_res ← ∅
          dist_res ← ∅
          lset ← lset ∩ v̄
          mset ← mset ∩ v̄
          my_work ← Redist(ToDo(total, res[p]),
                           lset ∩ mset, p)
          if mset ≠ v̄ and Min(mset) = p then
            mcast(⟨UPDATE, ⋃_{r∈v̄}(res[p] − res[r]), lset⟩)

        dlvr(⟨UPDATE, rset, pset⟩):
          foreach q ∈ pset do
            res[q] ← res[q] ∪ rset
          dist_res ← dist_res ∪ rset
          lset ← lset ∪ pset
          if p ∈ pset then
            mset ← view_comp
          if mset = lset = view_comp then
            my_work ← Redist(ToDo(total,
                             res[p] ∪ dist_res), view_comp, p)

        dlvr(⟨RESULT, w, r⟩):
          new_res ← new_res ∪ {(w, r)}

      esac
    od

    || Task 2:
    ComputeResults()
  coend

  function ToDo(T, C)
    return {w | w ∈ T ∧ ∄r : (w, r) ∈ C}
```

Figure 4. Partitionable Parallel Computation (Part a).

```
procedure ComputeResults()
  R ← new_res ∪ dist_res
  while ToDo(total, res[p] ∪ S) ≠ ∅ do
    w ← Min(ToDo(my_work, res[p] ∪ S))
    my_work ← my_work − {w}
    if w ≠ null then
      mcast(⟨RESULT, w, PerformWork(w)⟩)
  od
  res[p] ← res[p] ∪ S
```

Figure 5. Partitionable Parallel Computation (Part b).

new membership. Otherwise, in case of further failures (a leader crashes or ends in a different partition before multicasting the merging message), the merging protocol may have to continue in subsequent views.

We conclude by sketching how our solution exploits properties of view synchrony. Property GM2 guarantees that the work is distributed among all workers within a given partition, and thus the maximum possible speedup is achieved. Properties GM5, VS1, VS2 and VS3 allow a worker p to reason globally about which workers know the same set of results known locally. In particular, Property VS1 guarantees that two workers surviving from a view to the same next view will deliver the same set of messages (thus, both the new results and the results exchanged during the merging protocol). Property VS7 ensures that each merging message will be delivered in the same view in which it has been multicast; this simplify the algorithm construction, as workers do not have to worry about obsolete messages multicast during previous views. Properties GM1, GM3 and VS6 guarantee that a worker will eventually deliver the results of the subcomputations assigned to other workers, or it will install a new view (and thus redistribute the work). Finally, Property VS4 guarantees that no spurious messages will be delivered (thus, no incorrect results will be considered).

6. Conclusions

Specifications for services in asynchronous distributed systems require a delicate balance between two conflicting goals: they must be strong enough to be useful and exclude trivial solutions, yet they must be weak enough to be implementable [3]. The support layer we have specified in this paper has benefited from reflection and several revisions over an extensive period. The main objective of the current work has been to argue that realistic and interesting partition-aware applications can indeed be developed with ease on top of such a layer. The question of how it can be implemented efficiently on top of a typical operating system and an unreliable datagram communication service is treated in another work [6].

Information about group compositions conveyed through views and view changes happens to be appropriate for achieving partition awareness in network applications. Through a single mechanism (view change), we are able to abstract a large number of environment characteristics resulting from complex scenarios due to administrative intervention, crashes, recoveries, partitionings and mergings. As described in Section 5.1, group membership services alone are sufficient for only the simplest of network applications. More realistic network applications typically require closer cooperation among their components through communication. For this purpose, we have extended PGMS with a communication service based on view synchrony. The strong semantics provided by view synchrony regarding the composition and installation of views, delivery of messages, and most importantly, the integration between them, allows sophisticated global reasoning to be accomplished through local information alone and without having to resort to complex communication protocols.

The partition-aware applications we have developed reconfigure themselves in order to provide the "best" QoS that is possible in each partition. Obviously, no support layer can accomplish miracles and guarantee services at constant QoS levels despite partitionings: for example, no solution to the service activator problem can guarantee the availability of the service in a partition that contains no servers at all. The partition-aware applications we have considered have the simplifying characteristic that it is always possible to restore a consistent global state after recoveries or mergings. In general, applications that admit conflicting operations (e.g., updates to replicated data) in concurrent partitions may require additional system support for restoring a meaningful global state (if at all possible) after recoveries and mergings [4].

Acknowledgments

We are grateful to Alberto Bartoli for his comments on an early draft of this work.

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*, Le Mont Saint-Michel, France, June 1993.

[2] Y. Amir, D. Breitgand, G. Chockler, and D. Dolev. Group communication as an infrastructure for distributed system management. In *Proceedings of the 3rd International Workshop on Services in Distributed and Networked Environments*, Macau, June 1996.

[3] E. Anceaume, B. Charron-Bost, P. Minet, and S. Toueg. On the formal specification of group membership services. Technical Report TR95-1534, Department of Computer Science, Cornell University, Aug. 1995.

[4] Ö. Babaoğlu, A. Bartoli, and G. Dini. Enriched view synchrony: A programming paradigm for partitionable asynchronous distributed systems. *IEEE Transactions on Computers*, 46(6):642–658, June 1997.

[5] Ö. Babaoğlu, R. Davoli, L. Giachini, and M. Baker. Relacs: A communications infrastructure for constructing reliable applications in large-scale distributed systems. In *Proceedings of the 28th Hawaii International Conference on System Sciences*, pages 612–621, Maui, Hawaii, Jan. 1995.

[6] Ö. Babaoğlu, R. Davoli, and A. Montresor. Group membership and view synchrony in partitionable asynchronous systems: Specification and algorithms. Technical Report UBLCS-98-1, Department of Computer Science, University of Bologna, Mar. 1998.

[7] K. Birman. The process group approach to reliable distributed computing. *Commun. ACM*, 36(12):36–53, Dec. 1993.

[8] F. Cosquer, P. Antunes, and P. Verissimo. Enhancing dependability of cooperative applications in partitionable environments. In *Proceedings of the 2nd European Dependable Computing Conference*, Taormina, Italy, Oct. 1996.

[9] F. Cosquer and P. Verissimo. Large scale distribution support for cooperative applications. In *Proceedings of the European Reasearch Seminar on Advances in Distributed Systems (ERSADS)*, L'Alpe d'Houez, France, Apr. 1995.

[10] D. Dolev, D. Malki, and R. Strong. A framework for partitionable membership service. Technical Report CS95-4, Institute of Computer Science, The Hebrew University of Jerusalem, 1995.

[11] R. Friedman and A. Vaysburd. Implementing replicated state machines over partitionable networks. Technical Report TR96-1581, Department of Computer Science, Cornell University, 1996.

[12] F. Kaashoek and A. Tanenbaum. Group communication in the Amoeba distributed operating system. In *Proceedings of the 12th IEEE Symposium on Reliable Distributed Systems*, pages 222–230, Arlington, TX, May 1991.

[13] I. Keidar and D. Dolev. Efficient message ordering in dynamic networks. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, Philadelphia, PA, May 1996.

[14] F. Mattern. Virtual time and global states of distributed systems. In M. C. et. al., editor, *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. North-Holland, Oct. 1989.

[15] L. Moser, Y. Amir, P. Melliar-Smith, and D. Agarwal. Extended virtual synchrony. In *Proceedings of the 14th International Conference on Distributed Computing Systems*, Poznan, Pol, June 1994.

[16] A. Ricciardi and K. Birman. Using process groups to implement failure detection in asynchronous environments. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing*, pages 341–352, Aug. 1991.

[17] R. van Renesse, K. Birman, R. Cooper, B. Glade, and P. Stephenson. The horus system. In K. Birman and R. van Renesse, editors, *Reliable Distributed Computing with the Isis Toolkit*, pages 133–147. IEEE Computer Society Press, 1993.