Reliable Communication Infrastructure for Adaptive Data Replication

Mouna Allani[1], Benoît Garbinato[1], Amirhossein Malekpour[2], and Fernando Pedone[2]

[1] University of Lausanne
[2] University of Lugano

Abstract. In this paper, we propose a data replication algorithm adaptive to unreliable environments. The data replication algorithm, named Adaptive Data Replication (ADR), has already an adaptiveness mechanism encapsulated in its dynamic *replica placement* strategy. Our extension of ADR to unreliable environments provides a data replication solution that is adaptive both in terms of replica placement and in terms of request routing. At the routing level, this solution takes the unreliability of the environment into account, in order to maximize reliable delivery of requests. At the replica placement level, the dynamically changing origin and frequency of read/write requests are analyzed, in order to define a set of replica that minimizes communication cost. Performance evaluation shows that this original combination of two adaptive strategies makes it possible to ensure high request delivery, while minimizing communication overhead in the system.

1 Adaptive Data Replication

Data replication is a well-known technique to increase data availability and load balancing. A data replication system can be characterized by two key policies: a *replica placement policy*, which determines how many replicas the scheme creates and where it places them, and a *replica consistency policy*, which determines the level of consistency the scheme ensures among replicas, e.g., eager consistency or lazy consistency. These policies are typically implemented on top of a communication substrate ensuring a set of properties necessary for the correctness of the data replication system. An example of communication substrate is the *group communication* abstraction [1,5,8]. In this case, the *group communication* offers a set of guaranties including adaptiveness to membership changes, message ordering, and multicast reliability. In this paper, we define as a communication substrate a routing mechanism adaptive to unreliable environment in order to use it as the basis for a replica placement solution. We are primarily concerned with replica placement; replica consistency is out of the scope of the paper.

Regarding the replica placement policy, various replica management schemes have been proposed, based on a fixed number of replicas placed in fixed locations [2,15,13]. This approach works well when the source and the frequency of read and write requests are known in advance and remain static during the execution, which then implies that clients accessing the replicas are themselves static and generate

R. Meersman, T. Dillon, P. Herrero (Eds.): OTM 2009, Part I, LNCS 5870, pp. 634–652, 2009.

a steady stream of requests. When the frequency and the source of requests are variable, however, the ability to dynamically create, move, and delete replicas is essential when it comes to devising efficient replication schemes.

In a dynamic distributed environment, replica placement significantly affects the overall performance of the replication scheme. For example, since reading a replica locally is faster and less costly than reading it remotely, a widely distributed replication scheme is particularly well suited in read-intensive environments. On the other hand, writing to a large number of replicas may be slow and increase communication costs. For this reason, a narrowly distributed replication scheme is more adequate in write-intensive environments. In addition, the occurrence of node and link failures further challenges the effectiveness and performance of the replication scheme, as it can radically compromise replica placement decisions made before the failures occurred. The problem of placing replicas in dynamic and unreliable distributed environments advocates integrating *adaptiveness* into the replication schemes.

To adapt to the dynamic behavior of the environment and the application access patterns, various solutions have been proposed in the literature [16,11,12,19]. Among these, the *Adaptive Data Replication* algorithm (ADR) described in [19] is particularly interesting, as it was shown to be *convergent-optimal* with respect to communication costs. That is, as soon as the read-write access pattern changes, ADR adapts its replication scheme to minimize the communication cost caused by the routing of access requests. Intuitively, ADR organizes replicas as a connected graph, known as the *replication scheme*, which expands or contracts as the read-write access pattern changes.

Unfortunately, this convergence towards optimality only holds under two strict conditions: (1) the network is organized as a tree—finding an optimal replication scheme was shown to be NP-complete for general topologies [20]—and (2) no process or link failures occur. Condition 1 implies that one must first build an overlay tree covering the network. Condition 2 implies that ADR ceases to work correctly as soon as a failure happens.[1] Indeed, unreliable links may cause requests to be lost, thus misleading the replica placement strategy of ADR, while node failures may break the connectivity of the replication scheme, an essential assumption for ADR to work. Conditions 1 and 2 make ADR unsuitable to unreliable large-scale distributed environments.

Contributions. In this paper, we propose an architecture that extends ADR to make it capable of dynamically reorganizing itself based on changes in the application access patterns, and on link and node failures. The new replica placement strategy relies on a specialized routing layer, which encapsulates our *adaptive request routing strategy*. The latter is based on a tree overlay that aims at maximizing the reliability of request routing, in spite of link and node failures. This tree, named the *Maximum Reliability Tree* (MRT), is a spanning tree containing the most reliable paths in the system [4].

[1] In [19], processes switch to a special failure mode until recovery occurs. As detailed in Section 6, this approach is quite different from ours.

Roadmap. The remainder of this paper is organized as follows. Section 2 formally defines our model, describes and motivates the problem solved in the paper, and sketches the architecture of our solution. Section 3 presents our *adaptive request routing* algorithm based on a spanning tree maximizing the reliability of communication paths, while Section 4 describes an extension of the *adaptive replica* algorithm defined in [19], which aims at minimizing communication costs given a read-write pattern. In Section 5, we evaluate the benefit of using our *adaptive request routing* solution in terms of performance and adaptiveness, when both the access pattern changes and failures occur. Finally, Section 6 puts the proposed approach into perspective by comparing it with the state of the art; Section 7 concludes the paper and discusses future work.

2 A Modular Approach to Adaptiveness

In this paper, we consider an asynchronous distributed system composed of processes (nodes) that communicate by message passing. Our model is probabilistic in the sense that processes may crash and links may lose messages with a certain probability. More formally, the tuple $S = (\Pi, \Lambda, C)$ completely defines the (unreliable) environment considered in this paper. With Π the set of processes and Λ is a set of bidirectional communication links. We only consider systems with a connected graph topology. Process crash probabilities and message loss probabilities are modeled as *failure configuration* C.

We then define object o as the data to replicate, while $R \subseteq \Pi$ denotes the replication scheme of o, i.e., the set of nodes holding a copy of o. Any request sent to R is either a read or a write operation. Given these definitions, our approach consists in addressing the two following questions.

Adaptive Replica Placement. Given a pattern of reads and writes to o, what nodes should be part of R in order to minimize the communication cost?

Adaptive Request Routing. Given some failure configuration C, how should read/write requests be routed to maximize reliable delivery, and thus provide the replica placement layer with accurate information?

2.1 Adaptiveness to Access Patterns

The main idea of the adaptive replica placement strategy, borrowed from [19], consists in having the replication scheme R evolve like an amoeba along the branches of some tree-based communication overlay. The replica placement is managed in a fully decentralized manner. Each process p_k in R analyzes its access pattern and independently decides to either:

1. **expand** R, by sending a copy of o to one of its neighbors in the tree that does not yet hold a replica;
2. **contract** R, by quitting the replication scheme and discarding its copy of o;
3. **switch** R, by moving o to one of its neighbors in the tree, in case p_k is the only node holding a copy of o.

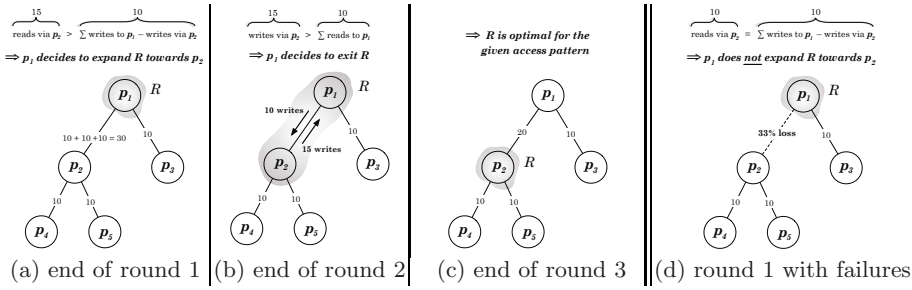


Fig. 1. Adaptive Replication Scheme – Example

Figures 1 (a) to (c) illustrate the behavior of the adaptive replica placement strategy on a concrete example, based on five processes. In this example, we initially have replication scheme $R = \{p_1\}$, i.e., p_1 is the only node with a copy of object o. In addition, we assume that each process periodically sends 5 reads and 5 writes to R and that they do so in synchronized rounds.[2] In the following, we define the communication cost of a round as the total number of requests transiting through any link in the system.

The situation after round 1 is shown in Figure 1 (a): p_1 received 15 reads and 15 writes from p_2 (5 reads and 5 writes from p_2 + 5 reads and 5 writes from p_4 + 5 reads and 5 writes from p_5), 5 reads and 5 writes via p_3 (5 reads and 5 writes from p_3 itself) and 5 reads and 5 writes from p_1 itself. p_1 notices that it received more reads via node p_2 (15 = 5 reads from p_2, 5 from p_4 and 5 from p_5) than the total number of writes originated from elsewhere, i.e., 10 writes (5 writes from p_1 + 5 from p_3). Based on this analysis, p_1 concludes that having a replica on node p_2 may improve the overall communication cost, which currently equals 60 (10 requests through link $l_{2,4}$ plus 10 through link $l_{2,5}$ plus 30 through link $l_{1,2}$ plus 10 through link $l_{1,3}$). This decision to expand R towards p_2 leads to the situation pictured in Figure 1 (b), with a communication cost of 55. To update all existing replicas, this change imposes however that p_1 and p_2 inform each other about the respective writes they received. At the end of round 2, p_1 finally decides to contract R by exiting the replication scheme. This situation is pictured in Figure 1 (c) and leads to a communication cost of 50. At this point, $R = \{p_2\}$ is the optimal replication scheme for the given access pattern.

2.2 Adaptiveness to Failures

To illustrate the need for adaptiveness to failures, let us revisit our example when injecting some unreliability into the system. As shown in Figure 1 (d), we inject a 33% message-loss into the link connecting p_1 and p_2, i.e., $l_{1,2}$ roughly loses one message out of three. So, at the end of the first round, p_1 compares the effective number of reads received via node p_2, which is equal to 10, with the number of all writes originated from elsewhere, which is also equal to 10. Based

[2] Synchronized rounds are only assumed to simplify our example and are by no means imposed by the replica placement strategy.

on this analysis, p_1 concludes that there is no need to expand R towards p_2 nor to switch with p_2, contrary to what happened in the setting shown in Figure 1 (a). That is, to process p_1, the replication scheme R appears to be optimal for the given access pattern, but this analysis is biased by the system unreliability, as the cost seen at p_1 is not the real cost imposed by the routing of requests. This observation clearly shows that in order to take full advantage of the adaptive replication scheme described earlier, we need to also adapt to the presence of failures when routing requests.

2.3 Solution Overview

Our solution follows the three-layer architecture pictured in Figure 2. The top layer executes the *adaptive replica placement* algorithm sketched earlier, which manages a replication scheme R changing according to the read-write pattern produced by some distributed application on top of it. The complete algorithm is described in Section 4.

The Replica Placement (RP) strategy relies on the *adaptive request routing* layer, which offers a set of communication primitives, and relies on a low-level *system layer* providing basic *best-effort* `send` and `receive` primitives. For the correctness of our solution, the best-effort aspect of the `send` primitive is hidden using a simple message resend/ack mechanism. For simplification, this retransmission mechanism was not included in the algorithm description. As detailed in Section 3, our routing solution permits to minimize the message overhead induced by this resend/ack mechanism by routing messages through the overlay including the most reliable paths covering the system: MRT. Thus, as shown in Section 5, our routing solution based on MRT induces an message overhead lower than when using any other tree overlay.

For each object o, the RP layer basically maps the corresponding replication scheme R to a dedicated group G managed by the Adaptive Routing (AR) layer. Coming to the AR layer, the latter offers the following adaptive and reliable services: (1) creation of a new group G and its announcement to all nodes in the system, (2) request routing from any node outside G to some node in G, and (3) multicasting among nodes in G. The algorithm executed by AR is detailed in Section 3.

Finally, the routing layer also relies on the system layer, not only for its send and receive primitives, but also for its ability to provide key information about nodes and links in the system. In particular, the system layer is responsible for providing an approximation of the failure rates of links and nodes, in terms of message-loss probabilities and crash probabilities respectively. That is, the system layer is capable of providing an approximation of the tuple $S = (\Pi, \Lambda, C)$ modeling the system.[3]

On Reliability and Consistency. Since our model is probabilistic, reliability should also be understood in probabilistic terms. Indeed, in the remainder of this paper, when we say for instance that our routing algorithm will *reliably* route a request, we actually mean that it will *maximize the probability* of the message

[3] In [4], we show how to use Bayesian statistical inference for this.

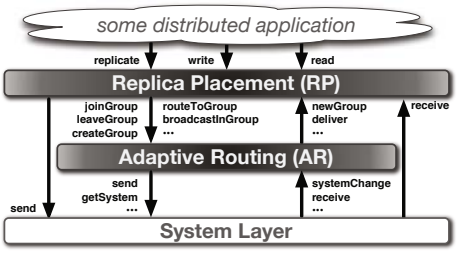


Fig. 2. Solution Architecture

reaching its destination, given the failure probabilities of system. Furthermore, as stated in Section 1, this paper is primarily focused on replica placement; replica consistency is out of its scope. As a consequence, the routing layer ensures no ordering guarantees on requests. Such a property if desired can be built on top of our system.

3 Adaptive Request Routing

For the sake of simplicity, we start by describing the lower layer of our solution: Adaptive Request Routing (AR). AR offers a reliable routing solution based on an underlying reliable overlay tree covering the whole system S named MRT. A group represents a connected subtree of this MRT. This group changes over time and our AR solution adapts accordingly. All group changes[4] are assumed to happen only at the group subtree leaves. That is, only a leaf of a group subtree can leave the group and a new node becomes a leaf of this subtree.

3.1 Interface

To create or change a group, AR provides the following primitives:

- *createGroup(gid)* enables a node to create a group of unique identifier *gid* and to be the first member of this group.
- *joinGroup(gid)* enables a node not in the group *gid* and with only **one neighbor in the group** to join this group and to be able to receive messages sent to the group members and to broadcast messages to the group members.
- *leaveGroup(gid)* called by a node in the group with only **one neighbor in the group**, it enables such a node to leave the group of id *gid* and thus stop receiving messages sent to the group.

For each group, once created, AR provides two services: (1) adaptive and reliable message routing from any node outside the group to some node in the group, (2) adaptive and reliable broadcasting among nodes in the group. These services are respectively encapsulated in the *routeToGroup(gid,m)* and *broadcastInGroup (gid,m)* primitives:

[4] By group change, we refer to an explicit leave/join to the group and not to node failure/recovery.

- *routeToGroup(gid,m)* enables a node to reliably send a message m to any member of the group *gid*.
- *broadcastInGroup(gid,m)* enables a node in group *gid* to reliably broadcast a message m to all members in the group.
- *deliver(gid, m)* works as a callback and enables a node to receive application message m.
- *newGroup(gid)* works as a callback and enables a node to receive an announcement of the new group *gid*.
- *resetGroup(gid)* works as a callback and enables a node to receive an announcement of a reset of the group *gid*, such a reset is due to an environment change.

The goal of AR is to take into account the environment unreliability by routing messages through a reliable tree named the Maximum Reliability Tree (MRT). It also adapts its communication services to group and environment changes.

3.2 Routing Algorithm

Algorithm 1 describes the main primitives provided by our communication layer. To create a group with a unique identifier *gid*, a process p_k calls the *createGroup (gid)* primitive. This primitive starts by announcing the new group to all members in the system. To do so, it broadcasts an initial message NEWGROUP through an underlying reliable tree overlay dedicated to this group. To that end, p_k first builds a tree *mrt* covering the whole system S using the *mrt*() primitive (line 9). This primitive is responsible for building the MRT with the root passed as argument (here p_k). If the *mrt* at p_k was already computed to serve other groups, p_k simply assigns it as the tree to serve group *gid* (line 10). The MRT of a process p_k contains the most reliable paths in S connecting p_k to all other processes in Π. Defined in [4] to ensure a reliable broadcast, MRT materializes the reliable aspect of our communication model. This paper does not detail the construction technique of MRT (see [4] for details). Then, p_k calls the *propagate()* primitive to launch the broadcast of the NEWGROUP (line 11). When a process p_k receives NEWGROUP message (line 15) from a neighbor p_j, it becomes aware of the group and knows how to route messages to it. To that end, p_k saves the tree overlay built to serve the new group *gid*: $T[gid]$ and a routing direction $direction[\ ]$ towards the group in $T[gid]$, which is the neighbor that forwards the NEWGROUP message to it (here p_j) (lines 16 & 17). Note that, at the source of the NEWGROUP message, this direction is set as the process itself as it represents the first member of the group (line 12). At the source and the receivers of the NEWGROUP message, a callback to the *newGroup()* primitive is performed to announce the new group to the upper layer (lines 14 & 18). Note that at any time, each member of a group *gid* knows all other members of this group. At the initialization process, the group consists only of the process that launched the NEWGROUP broadcast (line 13). Such a knowledge will be updated as soon as members start to join or leave the group.

A process p_k not in the group *gid* aiming at routing a message to any member of *gid* calls the *routeToGroup*() . This primitive simply sends the indicated message to the p_k's direction to the aimed group: $direction[gid]$ (line 24).

```
1: initialization:
2:    S ← getSystem()
3:    T ← ∅                                              {set of overlay trees}
4:    mrt ← ⊥
5:    direction ← ∅
6:    group ← ∅

7: procedure createGroup(gid)
8:    if mrt = ⊥ then
9:       mrt ← mrt(S, {p_k})
10:   T[gid] ← mrt
11:   propagate(T[gid], p_k, gid, NEWGROUP)
12:   direction[gid] ← p_k                               {root of gid is p_k}
13:   group[gid] ← {p_k}
14:   newGroup(gid)

15: upon receive(T_i, NEWGROUP, gid) via p_j do
16:   T[gid] ← T_i
17:   direction[gid] ← p_j
18:   newGroup(gid)
19:   propagate(T_i, p_j, gid, NEWGROUP)

20: procedure broadcastInGroup(gid,m)
21:   broadcast(gid, m)
22:   deliver(gid,m)

23: procedure routeToGroup(gid, m)
24:   send(gid, m) to direction[gid]

25: procedure joinGroup(gid)
26:   p_j ← direction[gid]
27:   send(FETCH-GROUP, gid) to p_j
28:   wait until receive(GROUP, gid, group_j) from p_j
29:   group[gid] ← group_j ∪ {p_k}
30:   broadcast(gid,JOIN)

31: upon receive(FETCH-GROUP, gid) from p_j do
32:   send(GROUP, gid, group[gid]) to p_j

33: upon receive(T_i, JOIN,gid) from p_i via p_j do
34:   group[gid] ← group[gid] ∪ {p_i}
35:   propagate(T_i, p_j, gid, JOIN)

36: procedure leaveGroup(gid)
37:   if direction[gid] = p_k then
38:      let p_j ∈ neighbors(gid) ∩ group[gid]
39:      send(NEWROOT, gid) to p_j
40:      direction[gid] ← p_j
41:   broadcast(gid,LEAVE)
42:   group[gid] ← ⊥

43: upon receive(NEWROOT, gid) do
44:   direction[gid] ← p_k

45: upon receive(T_i, LEAVE,gid) from p_i via p_j do
46:   group[oid] ← group[oid] \ {p_i}
47:   propagate(T_i, p_j, gid, LEAVE)

48: function group(gid)
49:   return group[gid]

50: function neighbors(gid)
51:   return {p_j : p_j ∈ V(T[gid]) ∧ l_{j,k} ∈ E(T[gid])}
```

Algorithm 1. Routing algorithm at p_k – Basic primitives

The second service provided by our communication layer is encapsulated in the *broadcastInGroup*() primitive. When called, this primitive calls the *broadcast()* primitive (line 21). The *broadcast()* primitive is called by a node in a group *gid* to broadcast a message among other members of *gid*.

To ensure an up-to-date knowledge about the group at each of its members, any join or leave event in this group is propagated to all members of the group. To join group *gid*, a node p_k calls the *joinGroup()* primitive. This primitive permits to p_k to obtain a view from the group it intends to join and to announce its arrival to other group members by calling *broadcast()* primitive to send a JOIN message to other members of the group.

To leave the group *gid*, a node p_k calls the *leaveGroup()* primitive. To ensure the correctness of the routing solution, each node p_k leaving a group *gid* has to check if it is the root of $T[gid]$ (line 37). If yes, p_k has to yield its root status to a neighbor in the group (line 38). Finally, p_k informs other members of the group about its leaving by sending a LEAVE message using the *broadcast()* primitive.

Algorithm 2 includes the *broadcast()* and *propagate()* primitives and callbacks delivering upper layer messages. The *broadcast()* primitive can be called by any member of a group *gid* to diffuse a message m to other members of the group. This primitive is used to diffuse both local messages to our communication layer (e.g., JOIN & LEAVE) and messages of the upper layer given by a call to the *broadcastInGroup()* primitive. This primitive first extracts T' as the subtree of $T[gid]$ covering the group (line 2). It then calls the *propagate()* primitive to send m through T' (line 3).

```
1:  procedure broadcast(gid, m)
2:     let T' ⊂ T[gid] : p_i ∈ T' ⇒ p_i ∈ group[gid]
3:     propagate(T', p_k, gid, m)

4:  procedure propagate(T', p_j, gid, m)
5:     for all p_i : link l_{k,i} ∈ E(T') ∧ j ≠ i do
6:        send(T', gid, m) to p_i

7:  upon receive(T', gid, m) from p_j do
8:     propagate(T', p_j, gid, m)
9:     deliver(gid, m)

10: upon receive(gid, m) from p_j do
11:    if (p_k ∈ group[gid]) then
12:       deliver(gid, m)
13:    else
14:       send(gid, m) to direction[gid]
```

Algorithm 2. Routing algorithm at p_k – Dissemination mechanism

3.3 Handling Failures and Configuration Changes

While being probabilistically reliable, the above communication algorithm does not ensure adaptiveness to the environment changes. Its adaptiveness is focused on group changes and on taking into account the environment unreliability. To enhance the adaptiveness of our communication solution, we extend AR as shown in Algorithm 3. This extension allows AR layer to adapt to failures that may

change the system topology and to take into account new configurations, i.e., new links and processes reliability. To adapt to environment changes, either regarding the topology or the configuration, we assume, as shown in Figure 2, that AR is notified by an underlying *System Layer*. The System Layer ensures at each process, the availability of an up-to-date view about the system. The details about how this layer obtains this view can be found in [4].

When a node p_k is notified by a new system view (line 1), if p_k is the root of at least one tree of one group (line 5), then p_k has the responsibility to define a new tree for that group in order to cover the new system configuration.

By building the new tree, p_k may break the connectivity property of the group members. To reconnect the group members in a subtree of $T[gid]$, p_k defines a new set of members of the group gid. This group is the set of nodes in the smallest subtree of $T[gid]$ including all old members of the group. Thus, some of the new group members were previously in the group, others are added by this mechanism only to heal the subtree connecting the group members. Then, p_k calls the *propagate()* primitive to disseminate the NEWTREE message annoucing the new tree $T[gid]$ and the new reconnected group to all nodes in S (line 9). When receiving a NEWTREE message from a neighbor p_j for a group gid, a process p_k assigns to its $T[gid]$ the given new tree T_{gid} (line 12) and changes its routing direction to p_j (line 13). Then, p_k checks if it was added to the new group while not being previously in the old one (lines 14 & 15). In this case, p_k calls the *forceJoin* primitive to inform the upper layer about this forced join. For this, p_k indicates p_j as the neighbor in the group. Note that p_j is in the group because it is the sender of the NEWTREE message indicating to p_k that it has to join the group. Thus p_j is either a previous member of the group or a new member forced in its turn to join the group in order to reconnect it. Section 4.1, details this further.

As a member of the group, p_k integrates the new group in its group view (line 17). Finally the source and the receivers of the NEWTREE message, call the *resetGroup()* primitive to announce the group change to the upper layer (lines 10 & 19).

```
1:  upon systemChange(S') do
2:    S ← S'
3:    if ∃ gid : direction[gid] = p_k then
4:      mrt ← mrt(S, {p_k})
5:      for all T[gid] ∈ T : direction[gid] = p_k do
6:        T[gid] ← mrt
7:        let T' be the smallest subree of T[gid] such that group[gid] ⊂ V(T')
8:        group[gip] ← V(T')
9:        propagate(T[gid], p_k, gid, NEWTREE, group[gid])
10:       resetGroup(gid)

11: upon receive(T_gid, NEWTREE, gid, group_gid) via p_j do
12:   T[gid] ← T_gid
13:   direction[gid] ← p_j
14:   if p_k ∈ group_gid then
15:     if p_k ∉ group[gid] then
16:       forceJoin(gid, p_j)
17:     group[gid] ← group_gid
18:   propagate(T_gid, p_j, gid, NEWTREE, group_gid)
19:   resetGroup(gid)
```

Algorithm 3. Routing algorithm at p_k – Adaptiveness to failures

Root failure. Note that in our AR solution, the root failure is problematic, as it represents the responsible for creating new covering tree if any changes happen. In addition, if such a root is a singleton, its failure results in the object being inaccessible. A solution for the singleton failure was proposed in [19], which we also retain in this paper. The idea is to impose a rule to the replica placement algorithm so that at any time at least two replicas of an object must be available.

When it comes to the root failure, several solutions were proposed in this context [6,9]. Similarly, we can replicate the root to improve its reliability. Details of this strategy could be found in [6].

4 Adaptive Replica Placement

The Replica Placement (RP) layer defines, for each object o to replicate, a replication scheme R changing according to the read-write pattern to o in order to move R towards the center of the read-write activity. When the read-write pattern is stable, R eventually converges towards the optimal replication scheme ensuring the minimum communication cost.

4.1 Initialization and Replica Access

The RP layer relies on the AR layer for each communication step. For each object o, AR layer manages a group that corresponds to the set of processes holding a replica of o, i.e., R. In other words, our replication scheme R at RP is seen at AR as a group of processes to which AR provides a set of communication services. RP refers to AR to get information about the neighborhood in the overlay defined by AR to serve the group of one object o by calling the *neighbors()* primitive. It also refers to AR to get a view of R using the *group()* primitive. To adapt the underlying communication solution to the R changes (i.e., to the group changes), RP informs AR about all changes in R by calling the *joinGroup*() and *leaveGroup*() primitives.

Algorithm 4 details the primitives provided by RP to any upper application. The management of replicas of an object o starts by a call to the *replicate*() primitive by the initial process p_k holding o. In this initialization step, p_k calls the *createGroup*() primitive (line 7) of the AR by indicating *oid* as the identifier of the object o for which p_k wants to create a replication scheme (or a group). As detailed in Section 3.2, the *createGroup*() primitive reliably announces the object o to all processes in the system and creates a group dedicated to this object.

The following replica placement steps (i.e., replica creation or replica discarding) are then performed cooperatively based on statistics collected locally at each process concerning the received requests. At a process p_k, for each object o of identifier *oid*, *reads[oid]* and *writes[oid]* respectively refer to the total number of reads and the total number of writes p_k received for the object o. These counters also include for each neighbor p_j of p_k (according to the overlay defined by AR to serve the group dedicated to o) the number of reads, *reads[oid, p_j]* and the number of writes, *writes[oid, p_j]* received from p_j for the object o.

```
 1: initialization:
 2:    reads ← ∅                                          {set of read counters}
 3:    writes ← ∅                                         {set of write counters}
 4:    Ω ← ∅                                              {set of local objects}
 5:    leavePending ← ∅                                   {set of boolean}

 6: procedure replicate(o)
 7:    createGroup(o.id)

 8: function read(oid) : state
 9:    if p_k ∈ group(oid) then
10:       return Ω[oid].state
11:    else
12:       routeToGroup(oid,READ)
13:       wait until receive(RESPONSE,o) with o.id = oid
14:       return o.state

15: procedure write(oid, state)
16:    if p_k ∈ group(oid) then
17:       broadcastInGroup(oid,WRITE,state)
18:    else
19:       routeToGroup(oid,WRITE,state)

20: upon newGroup(oid) ∨ resetGroup(oid) do
21:    for all p_j ∈ neighbors(oid) do
22:       reads[oid, p_j] ← 0
23:       writes[oid, p_j] ← 0
24:    leavePending[oid] ← false

25: upon forceJoin(oid, p_j) do
26:    send(FETCH-OBJECT, oid) to p_j
27:    wait until receive(OBJECT, o) from p_j with o.id = oid
28:    Ω[oid] ← o

29: upon receive (FETCH-OBJECT, oid) from p_j do
30:    send(OBJECT, Ω[oid]) to p_j

31: upon deliver(oid,READ) from p_i via p_j ∈ neighbors(oid) do
32:    send (RESPONSE,Ω[oid]) to p_i
33:    reads[oid] ← reads[oid] + 1
34:    reads[oid, p_j] ← reads[oid, p_j] + 1

35: upon deliver(oid,WRITE,state) via p_j ∈ neighbors(oid) ∪ {p_k} do
36:    if p_j ∉ group(oid) then
37:       broadcastInGroup(oid,WRITE,state)
38:    else
39:       Ω[oid].state = state
40:       writes[oid] ← writes[oid] + 1
41:       writes[oid, p_j] ← writes[oid, p_j] + 1
```

Algorithm 4. Adaptive replication at p_k – Reading & Writing

When notified of a new group (line 20), process p_k becomes aware of the replication scheme R of the corresponding object o and initializes its counters *reads[ ]* and *writes[ ]* according to its set of neighbors defined by AR (lines 22 & 23).

To read or write a state at o, p_k calls, respectively, the *read()* and *write()* primitive. If p_k has a replica of o, i.e., it is a member of the group dedicated to o (lines 9 & 16), the *read()* function simply returns the state of the replica extracted from the local structure Ω. The *write()* function, in this case, calls the *broadcastInGroup*() primitive (line 37) of the AR algorithm to broadcast the update within processes of the replication scheme of o. Otherwise, these primitives call the *routeToGroup*() primitive to route the request to a process holding a replica of o.

When a process p_k receives a write or a read request for *oid* (lines 35 & 31), it respectively updates its local replica (line 39) or sends back the response extracted from its replica (line 32), then updates its counters accordingly (lines 40 & 41 - 33 & 34).

To suport the adaptiveness to environment changes, our RP algorithm provides the *forceJoin()* primitive (line 25). This primitive permits to an underlying

```
1: periodically do :
2:    for all oid ∈ Ω do
3:       if p_k ∈ group(oid) then
4:          if ¬ tryExpanding(oid) then
5:             if group(oid) = {p_k} then
6:                trySwitching(oid)
7:             else if | neighbors(oid) ∩ group(oid) | = 1 then
8:                tryContracting(oid)
9:          reads[oid] ← 0
10:         writes[oid] ← 0

11: function tryExpanding(oid) : boolean
12:    success ← false
13:    candidates ← neighbors(oid) \ group(oid)
14:    for all p_j ∈ candidates do
15:       if reads[oid, p_j] > writes[oid] − writes[oid, p_j] then
16:          send(JOIN,Ω[oid]) to p_j
17:          success ← true
18:    return success

19: upon receive (JOIN,o) do
20:    Ω[o.id] ← o
21:    joinGroup(o.id)                          {inform AR about the group change}

22: procedure trySwitching(oid)
23:    if ∃ p_j ∈ neighbors(oid) : 2 × (reads[oid, p_j] + writes[oid, p_j]) > reads[oid] + writes[oid]
       then
24:       send(BE-SINGLETON,Ω[oid]) to p_j
25:       wait until receive(ACK-SINGLETON,oid) from p_j
26:       leaveGroup(oid)                       {inform AR about the group change}
27:       Ω[oid] ←⊥

28: upon receive (BE-SINGLETON,o) do
29:    Ω[o.id] ← o
30:    joinGroup(o.id)                          {inform AR about the group change}
31:    send(ACK-SINGLETON,o.id)

32: procedure tryContracting(oid)
33:    let p_j ∈ neighbors(oid) ∩ group(oid)
34:    if writes[oid, p_j] > reads[oid] then
35:       leavePending[oid] ← true                    {Trying to contract from oid}
36:       send(REQUEST-LEAVE,oid) to p_j
37:       wait until receive(REPLY-LEAVE,reply,oid) from p_j
38:       if reply then
39:          leaveGroup(oid)                    {inform AR about the group change}
40:          Ω[oid] ←⊥
41:       leavePending[oid] ← false

42: upon receive(REQUEST-LEAVE,oid) from p_j do
43:    if ¬leavePending[oid] then
44:       send(REPLY-LEAVE,true,oid)                  {Not in contract test from oid}
45:    else
46:       reply ← p_k > p_j
47:       send(REPLY-LEAVE,reply,oid)
```

Algorithm 5. Adaptive replication at process p_k – Replicas Placement

communication layer to add a member to one replication scheme R. When called at a node p_k, this primitive fetches a copy of the object with the indicated unique identifier from the indicating node p_j as a neighbor in R. It then includes o to become a member of its replication scheme (line 28).

4.2 Adaptive Placement

As soon as the requests for object o start to be submitted, the RP solution adapts the replication scheme R to the read-write pattern in a decentralized manner. Starting as a singleton, R may expand (by placing new replicas in appropriate processes), switch (by changing the replica holder if R is a singleton) or contract (by retrieving replicas from a specific process, if R is not a singleton) while remaining connected. These actions are tested periodically by some processes in R based on a set of statistics concerning the received requests. Algorithm 5 gives a formalization of the adaptive replica placement detailed in [19]. Hereafter we describe this algorithm and its interaction with AR layer.

The expansion test is executed by each process p_k in R (line 3) with at least one neighbor not in R (line 13). To do so, for each neighbor p_j not in R (line 14), p_k sends a 'Join' request to p_j (line 16) if the number of reads that p_k received from p_j is greater than the total number of writes received from elsewhere (line 15). When a node p_k joins R, it informs AR about the new member of the group by calling the *joinGroup*() primitive (line 21).

If the expansion test fails (line 4) and p_k is the singleton (line 5), it executes the switch test (line 6). To switch, p_k should first find a neighbor p_j such that the number of requests received by p_k from p_j is greater than the number of all other requests received by p_k (line 23). Then p_k sends a copy of the object o to p_j with an indication that p_j becomes the new singleton of R (line 24). Before discarding its local replica (line 27) p_k must receive a confirmation from p_j to ensure a non empty replication scheme (line 25). To inform AR, p_k and p_j respectively call the *leaveGroup*() and *joinGroup*() primitives (lines 26 and 30).

The contraction test is also executed after a failed expansion test but when p_k has only one neighbor in R: p_j (line 7). To contract from R, p_k should have more writes from p_j than all received reads (line 34), In this case, p_k requests permission from p_j to leave R (line 36). If permitted (line 38), p_k discards its local replica (line 40) and informs AR (line 39). The permission request sent to p_j permits to manage a possible risk of mutual contractions. If not managed, such a risk may induce an empty replication scheme when p_k and p_j constitute all elements of R. When detected (line 45) such a conflict is resolved by a simple mechanism permitting to the process with the lower id to leave R (line 46).

5 Evaluation

The efficiency of the adaptive replica placement algorithm has been previously proved in [19]. In this section we are more concerned by the immersion of this algorithm in unreliable environment. That is, we aim at evaluating the advantage of using AR (based on MRT as the network overlay) as the underlying communication model of the RP solution.

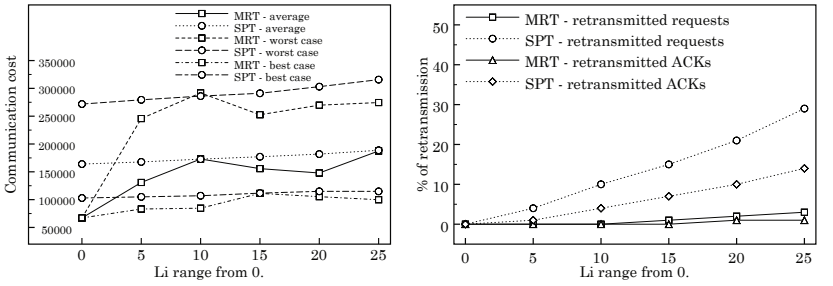
Evaluation method. To evaluate the advantage of our reliable communication model, we define a comparison tree to MRT as a covering tree built independently from links reliability. Such a tree could be the minimum latency tree defined in [19]. We name this comparison tree SPT. SPT is any covering tree built without taking into account the reliability of the system components (links & processes). To compare the impact of MRT and SPT in our data replication solution, we define an evaluation method that consists of measuring the cost needed to completely hide the impact of the system unreliability. Indeed, to converge towards the same replication scheme as in a reliable environment, a simple idea is to resend each request until receiving an acknowledgement (ACK) for it. The number of retransmissions depends on the reliability of the link through which the request should be sent. In a reliable link, an ACK is received after the first request, in which case no retransmission is needed.

To show the benefit of using our MRT as the overlay of our communication model, we compare the retransmission needed when using both MRT and SPT to hide the environment unreliability. This comparison regards the communication cost, the number of messages retransmitted and its corresponding percentage as the portion of sent messages that represents the retransmissions.

Simulation configuration. To evaluate our solution, we conducted a set of experiments for various network configurations with 100 processes connected randomly with an average connectivity of 8, i.e., 8 direct neighbors. To simplify our results interpretation, we varied the links configuration L_i while assuming that processes are reliable i.e, $\forall p_i : P_i = 0$. At each experiment, we simulate the replication of one object and we assign to each process a fixed number of read and write requests that it submits periodically. The initial copy of the object to replicate is held by the node that initiates the replication, which is chosen randomly among the system nodes. We then measured the number of propagated messages (initial messages and retransmitted messages) at the convergence, i.e., when R stabilizes. In our simulation, we also assume that ACK messages are subject to loss.

5.1 AR Benefit

Figure 1 (a) shows the communication cost (in terms of all routed messages) of our solution when using both MRT and SPT at the convergence. The communication cost includes the number of requests submitted and the number of retransmitted messages (request or ACK) necessary to hide the unreliability. In Figure 1 (a), we show the average communication cost over executions, the worst and the best cases that our executions sample detected. In the corresponding executions we assigned, at each process, a periodic number of *Reads* in [0, 50] and a periodic number $Writes$ in [0, 50]. When using MRT our solution induces a lower communication cost. Globally this cost increases as the link unreliability increases. This shows that as the reliability worsens, more message retransmission is needed. Note that the number of original requests (not including retransmissions) to route at the convergence is different for MRT and SPT since the generated replication scheme R is different. Indeed, the form of the tree influences significantly the resulting replication scheme as each tree has a



(a) All routed messages (b) % of retransmitted ACKs & requests

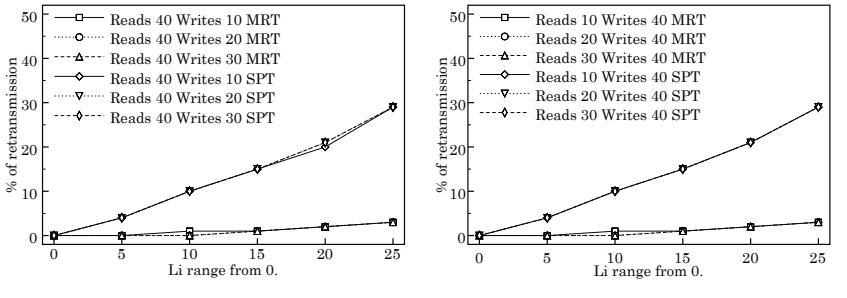
Fig. 1. Communication cost

different origin of requests reaching its replication scheme. For this reason, hereafter, we focus our evaluations on the induced retransmission to hide the system unreliability.

Figure 1 (b) shows the percentage of request messages and ACK messages retransmitted to ensure that every inter-nodes message (e.g., forwarded requests) is received when using the MRT and SPT trees. As shown in Figure 1 (b), this percentage is lower when using MRT than when using SPT. This difference increases as the links unreliability is in a larger range to reach the 15% when the links unreliability L_i is in [0 - 25%].

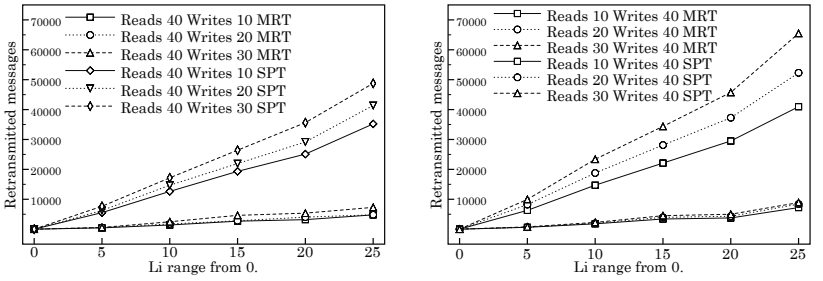
5.2 Varying Read/Write Pattern

In this section, we vary the range of fixed *Reads* and *Writes* assigned to nodes in order to evaluate their impacts on the retransmission cost. The indicated values of *Reads* and *Writes* in figures below represent the lower bound of a range of size 10. Figure 2 shows the percentage of requests retransmission needed to hide the system unreliability using MRT and SPT. As noticeable the variation of the Read/Write pattern has no impact on the percentage of retransmitted requests.



Fixed Reads > Writes Fixed Writes > Reads

Fig. 2. % of retransmitted requests to hide unreliability



(a) Fixed Reads > Writes (b) Fixed Writes > Reads

Fig. 3. Number of retransmitted messages

This is however not the case for their corresponding number of retransmissions shown in Figure 3. The number of retransmissions needed to completely hide the links unreliability when using SPT is much higher than the one needed when using MRT. This difference increases as the links unreliability increases to reach 7 times more retransmissions when the links unreliability is in [0 - 25%]. Using both overlays structure, the number of retransmissions increases as the number of requests ($Reads + Writes$) increases. For the same fixed number of periodic $Reads$ at each process, when the number of periodic $Writes$ increases it induces more messages retransmissions than when we increase the number of $Reads$ while fixing the number of $Writes$. This is due to a different replication scheme at the convergence. In Figure 3 (a), we have more $Reads$ than $Writes$, which implies a large replication scheme through which each submitted $Write$ is broadcast. The larger the replication scheme, the higher is the cost induced by the broadcast of $Writes$. And as the global number of $Writes$ increases, this cost increases. In Figure 3 (b), we have more $Writes$ than $Reads$, which implies a small replication scheme. For the same number of global $Writes$, when the number of $Reads$ increases the number of propagated messages increases by the propagation of each $Read$ through a tree branch until reaching the replication scheme. This is less costly than a broadcast through a large replication scheme.

6 Related Work

Research efforts have considered distributed adaptive data replication systems from many different angles. Many previous works on adaptive data replication have considered user requests of some sort as the parameter to adapt to. In doing so, most of them formalize the problem as a cost function to optimize. The precise interpretation of adaptive data replication depends on the cost function. In [19], the replica management adapts to the read-write pattern. Based on the approach defined in [19], in this paper, our objective is to adapt to the read-write pattern while coping with an unreliable environment. While handling process and link failure-recovery the approach proposed [19] does not take into

account the environment unreliability in its communication model. In [19], the link or process failure is handled by switching the execution in a failure mode where the replication scheme of each object is a singleton named *primary processor*. The execution then returns to the normal mode when the failed component (link or process) recovers. In our paper, however, the environment unreliability is adressed in a preventive way since our communication model selects apriori the most reliable paths to serve the requests routing in order to minimize the component failures risk. In addition, when a failure happens, our replication scheme is simply reconnected instead of being reduced to a singleton. The environment unreliability in this context was adressed also in a preventive way in [10] but in a different manner than ours. The approach defined in [10], distributes replicas in locations (or servers) whose failures are not correlated in order to mitigate the impact of correlated, shared-component failures. The adaptiveness of this approach relies on the proposed placement strategy as the number of replicas to place is assumed to be fixed by the storage system. Similarly to our paper, some works [17,18] integrate a communication model for their adaptive data-replication solutions. Other works have also adapted to the read-write pattern, e.g., in [7,16]. In addition to the read-write pattern, several objectives were defined to dictate the replication strategy. The approach defined in [7] also takes into consideration storage costs and node capacity to serve requests. In [11], a protocol dynamically replicates data so as to improve the client-server proximity without overloading any of the replica holder. In [3], the replica allocation aims to balance loads in terms of CPU and disk utilization in order to increase the system throughput. In [14], the replica placement strategy aims to minimize a cost function taking into account the average read latency, the average write latency and the amount of bandwidth used for consistency enforcement. Contrary to the distributed definition of the replication scheme of this paper, in [14] the definition of the placement configuration is done at a central server (the origin server) evaluating the cost of different possible configurations to select the best placement configuration yielding the least cost. Similarly, in [12] the placement configuration changes are decided at a central site based on statistics about the accesses data measured in a distributed manner. The starting point in [12] is also different than the one defined in this paper. That is, in [12], at system start-up, a full copy is available at each edge server. Throughout execution, a self-optimization algorithm is triggered periodically.

7 Conclusion

This paper proposed an adaptive replica placement solution for unreliable environments. The adaptiveness of this solution is at the replica placement level and the request routing level. In our evaluation, we showed that the unreliability impact on our replica placement algorithm could be hidden with a minor cost using our reliable communication model. However real-world deployment in WAN suffers from other contraints. One major constraint to be considered is the limited memory and CPU power at processes. Thus, assuming that each process has a global knowledge prevents this solution to scale in such environment.

References

1. Amir, Y., Tutu, C.: From total order to database replication. In: Proceedings of ICDCS, pp. 494–503. IEEE, Los Alamitos (2002)
2. Bernstein, P.A., Goodman, N., Wong, E., Reeve, C.L., Rothnie Jr., J.B.: Query processing in a system for distributed databases (sdd-1). ACM Trans. Database Syst. 6(4) (1981)
3. Elnikety, S., Dropsho, S.G., Zwaenepoel, W.: Tashkent+: memory-aware load balancing and update filtering in replicated databases. In: Euro. Sys., pp. 399–412 (2007)
4. Garbinato, B., Pedone, F., Schmidt, R.: An adaptive algorithm for efficient message diffusion in unreliable environments. In: Proceedings of IEEE DSN (2004)
5. Holliday, J., Agrawal, D., El Abbadi, A.: The performance of database replication with group multicast. In: Proceedings of FTCS, pp. 158–165. IEEE Computer Society Press, Los Alamitos (1999)
6. Jannotti, J., Gifford, D.K., Johnson, K.L., Kaashoek, M.F., O'Toole Jr., J.W.: Overcast: Reliable multicasting with an overlay network. In: Proceedings of OSDI (October 2000)
7. Kalpakis, K., Dasgupta, K., Wolfson, O.: Optimal placement of replicas in trees with read, write, and storage costs. IEEE Trans. Parallel Distrib. Syst. 12(6) (2001)
8. Kemme, B., Bartoli, A., Babaoglu, Ö.: Online reconfiguration in replicated databases based on group communication. In: DSN, pp. 117–130 (2001)
9. Kostic, D., Rodriguez, A., Albrecht, J., Bhirud, A., Vahdat, A.: Using random subsets to build scalable network services. In: Proceedings of USITS (March 2003)
10. MacCormick, J., Murphy, N., Ramasubramanian, V., Wieder, U., Yang, J., Zhou, L.: Kinesis: A new approach to replica placement in distributed storage systems. ACM Transactions on Storage (TOS) (to appear)
11. Rabinovich, M., Rabinovich, I., Rajaraman, R., Aggarwal, A.: A dynamic object replication and migration protocol for an internet hosting service. In: ICDCS (1999)
12. Serrano, D., no-Martínez, M., Jiménez-Peris, P.R., Kemme, B.: An autonomic approach for replication of internet-based services. In: SRDS, Washington, DC, USA, pp. 127–136. IEEE Computer Society, Los Alamitos (2008)
13. Serrano, D., Patiño-Martínez, M., Jiménez-Peris, R., Kemme, B.: Boosting database replication scalability through partial replication and 1-copy-snapshot-isolation. In: PRDC, pp. 290–297 (2007)
14. Sivasubramanian, S., Alonso, G., Pierre, G., van Steen, M.: GlobeDB: Autonomic data replication for web applications. In: Proc. of the 14th International World-Wide Web Conference, Chiba, Japan, pp. 33–42 (May 2005)
15. Stonebraker, M.: The design and implementation of distributed ingres. In: The INGRES Papers (1986)
16. Tsoumakos, D., Roussopoulos, N.: An adaptive probabilistic replication method for unstructured p2p networks. In: OTM Conferences, vol. (1) (2006)
17. van Renesse, R., Birman, K.P., Hayden, M., Vaysburd, A., Karr, D.A.: Building adaptive systems using ensemble. Softw., Pract. Exper. 28(9) (1998)
18. Vaysburd, A., Birman, K.P.: The maestro approach to building reliable interoperable distributed applications with multiple execution styles. TAPOS 4(2) (1998)
19. Wolfson, O., Jajodia, S., Huang, Y.: An adaptive data replication algorithm. ACM Trans. Database Syst. 22(2) (1997)
20. Wolfson, O., Milo, A.: The multicast policy and its relationship to replicated data placement. ACM Trans. Database Syst. 16(1) (1991)