TRANSPARENT DATA RELOCATION IN HIGHLY AVAILABLE DISTRIBUTED SYSTEMS

SPYROS VOULGARIS, MAARTEN VAN STEEN, ALINE BAGGIO, AND GERCO BALLINTIJN

Abstract. In a distributed system, long-running distributed services are often confronted with changes to the configuration of the underlying hardware. If the service is required to be highly available, the service needs to deal with the problem of adapting to these changes while still providing its service. This problem is increased further if multiple changes can occur concurrently. In this paper, we describe a method that solves this problem by carefully shipping data and forwarding requests to appropriate hosts. Our method specifically enables the distributed service to deal with concurrent changes in a concurrent fashion, thereby promoting the efficiency of the service.

1. Introduction

A service in a distributed system is often implemented as a set of cooperating server processes distributed among multiple machines. These server processes handle requests from client processes and jointly manage the data and computations that comprise the (distributed) service. Managing such a set of servers has many facets. In this paper we focus on one facet namely the problem of redistributing data between the server processes. This problem is usually considered part of configuration management [1].

In a distributed system, server processes are frequently added, moved, or removed, for instance, to adapt the system to changes in its usage. As a result of these configuration changes, the data stored at the servers needs to be redistributed to reflect the updated set of servers. Changes to the configuration of the system are not the only reason for data redistribution among servers. For instance, a change in the load distribution policy used by the service (e.g., the introduction of a new load balancing scheme), would also result in the redistribution of data.

Ideally, data redistribution should be done in a way that is transparent to client processes. To accomplish this transparency, we need to solve two problems: (1) how to locate data, and (2) how to move data while allowing operations on that data to be processed. Much work has been done on locating mobile data, or more generally, objects [2]. However, mechanisms for handling mobility only partly solve our problem as we also need to guarantee continuous access to the data that is being moved.

In this paper we describe a solution for achieving such transparency for distributed services. The main contribution of this paper is that we show how data redistribution can take place in a distributed system in a way that is transparent to clients. Our solution specifically enables the service to continue to operate, and thus does not compromise its availability.

The remainder of this paper is organized as follows. We continue in Section 2 with a description of the model of the distributed system we want to support. In Section 3, we describe the basic structure of our solution when applied to a single configuration change. Afterward, in Section 4, we explain some of the design alternatives we have considered. In Section 5, we extend our solution and examine several ways in which it can deal with concurrent configuration changes. We describe some related work in Section 6 and conclude with Section 7.

2. The System Model

In our system model, the data that is managed by a distributed service consists of a (potentially large) set of self-contained *data records*, or simply records. Every record in the service has its own *identifier* and *value*, which can be read or written. Over the course of time, new records will be added to the service and existing ones will be removed. The service is implemented by a set of *server processes*, with each server located on a different machine and managing (or hosting) a disjoint subset of the records. Every record is always hosted by a single server, which we call the record's *hosting server*. In other words, we assume that data records are not replicated. Records are assigned to servers based on a deterministic load distribution policy that can change over time.

The distributed service provides its services to external *client processes*. A client submits a request to perform an operation on a record by providing the type of operation, the identifier of the record, and the set of parameters for the operation. We distinguish two kinds of operations: *lookup* and *update* operations. A lookup operation is read-only; it returns the value of a record without modifying its content. In contrast, update operations include all operations that either change the value of a record, or that add or remove a record from the current set of records.

The load distribution policy is captured by a data structure which we call the *mapping*. This data structure defines for each record its hosting server. For easy access, each server keeps a local copy of the mapping. To invoke an operation, a client arbitrarily picks a server and submits its request to it. The selected server then looks in its copy of the mapping to determine the record's hosting server and forwards the request accordingly. We assume reliable communication for both client-server and server-server request passing. As we shall see later, a server may keep copies of more than one mapping when configuration changes are in progress. In such a case, the oldest mapping is called the *authoritative mapping* and is used to forward requests.

Whenever a server is added or removed from the distributed service or when its load distribution policy is changed, the placement of the records at the servers may no longer adhere to the load distribution policy. If that is the case, the service needs to redistribute its records over its servers. We refer to this process as *record relocation*, or simply relocation. As a result of this relocation, the mapping needs to be updated to reflect the new distribution of the records over the servers.

Mappings are managed by a separate service, called the *configuration service*. Whenever the configuration service is informed about a configuration change, it is responsible for building an updated mapping that includes the configuration change and for providing the new mapping to all the servers. When a server receives the new mapping, it starts relocating records. Transferring of mappings between a server and the configuration service, as well as relocation of records between servers are carried out by means of reliable communication. The internal design of the configuration service is out of the scope of this paper.

It is important that the update of the mapping and the subsequent relocation of records are transparent to client processes. The problem we are thus faced with is *how* to relocate records in the distributed service while still guaranteeing continuous availability of the records to the clients. Clients should be able to simply submit a request for any record at any times, that is, before, during and after the relocation of the record. Note that we also like the solution to complete the configuration change in a timely manner. We do not consider security issues in this paper and assume that servers and communication can be trusted.

3. The Solution for a Single Redistribution

Our solution for the transparent redistribution of records consists of the following three steps:

Initialization: Initially, all the servers have a local copy of the authoritative mapping M, which is used for forwarding requests to the proper hosting servers of the records involved. When the configuration service receives the notification for a configuration change, it computes a new mapping M' that reflects the change, and distributes M' to *all* the servers of the distributed service.

Record relocation: When a server receives a new mapping M', it checks if some of its own records have to be relocated, and ships (relocates) the records remapped by M' to their respective new hosting servers.

During the record relocation step, servers continue to forward client requests using the authoritative mapping M. A server can, therefore, be handed a request for a record that it should host under M, but that is remapped by M'. Requests involving such an *already-shipped* record are *forwarded* to the record's *new* hosting server as dictated by M'. In this way, the authoritative hosting server acts as a proxy for the already-shipped records. A request involving a *not-yet-shipped* record is simply serviced locally by the *authoritative* hosting server, that is, the server as dictated by the authoritative mapping M.

Termination: As soon as a server completes its record relocation step, it notifies the configuration service. When the configuration service receives completion notifications from *all* servers, it, in turn, notifies all the servers that the termination step can start. During the termination step, each server simply discards mapping M and replaces it by M', which then becomes the new authoritative mapping.

Once the termination step is over, servers that are destined to be removed are free to shutdown, and newly added ones can expect to be handed requests directly for records they host. Variations in the delivery time of messages from the configuration service to different servers may cause a temporary inconsistency between some servers where some may still regard M as authoritative while others are already using M'. As a consequence, if a terminating server learns about the completion of the configuration change and shuts down before some other server has been notified, the latter may still attempt to forward a request to the then terminated server. Should such a situation occur, the forwarding server contacts the configuration service to be updated on the authoritative mapping and forward the request to the new authoritative hosting server.

To make the record relocation step more efficient, a server does not discard a record after it has been shipped to its new host. Instead, the server keeps handling *lookup* requests for such a record, but only for as long as that record remains consistent with the copy at its new hosting server. An already-shipped record is considered consistent with its copy at the new hosting server until the first *update* request for that record is made. After the first update request is

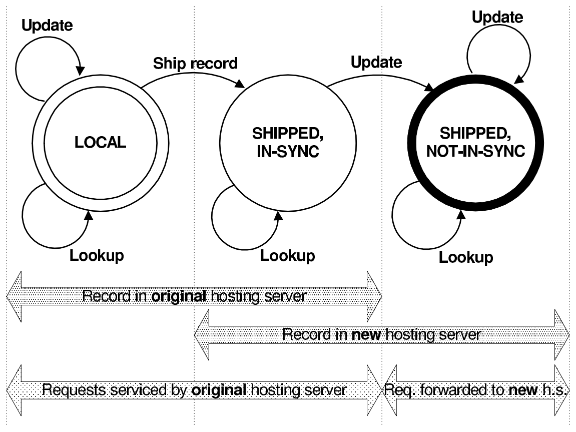
Update
Ship record
Update
Update
LOCAL
SHIPPED, IN-SYNC
SHIPPED, NOT-IN-SYNC
Lookup
Lookup
Lookup
Record in **original** hosting server
Record in **new** hosting server
Requests serviced by **original** hosting server
Req. forwarded to **new** h.s.

Figure 3.1: State diagram for relocating records

received for an already-shipped record, the server forwards this and *all* subsequent requests (including lookups) for this record to the record's new hosting server.

A request for a record that can be handled at its current hosting server, even if it has already been shipped, is referred to as a *locally serviceable request*. Note that until the relocation step is over, servers forward client requests using the current authoritative mapping. Hence, all requests for a record will be forwarded to its current hosting server, even when the record has been shipped to its new hosting server.

To ensure consistency, a server associates a *state* flag with each of its records. Figure 3.1 shows the state diagram that controls the behavior of a server with respect to a record. All records are initially assigned the *LOCAL* state and requests for them are handled locally. When a record is shipped to its new hosting server, its state changes from *LOCAL* to *SHIPPED IN-SYNC*. Lookup requests for that record will continue to be handled locally until the first update request arrives. When the authoritative hosting server receives the first update request for that record, its state changes from *SHIPPED IN-SYNC* to *SHIPPED NOT-IN-SYNC*. From that point on, the authoritative hosting server delegates responsibility for that record to the new hosting server by forwarding it all requests for that record. The authoritative hosting server is now free to remove the record from its local storage.

4. Alternative Design Considerations

A number of design decisions were taken in our solution on how to carry out the tasks associated with the redistribution of data. For some of these tasks, alternative strategies could have been employed. For instance, the relocation of a record to its new hosting server and the handling of client requests during a redistribution could be done differently. In this section, we present alternative approaches to these tasks and motivate our choices.

First consider the relocation of records. In our solution, a record that needs to be relocated is *pushed* by its authoritative hosting server to its new hosting server. An alternative is to let records be *pulled on-demand* by their respective new hosting servers. In this alternative, the new mapping M' is distributed to all the servers, but no record shipping starts. Instead, when a server receives a request for a record it *should* — but does not yet — have, it fetches the record from its authoritative hosting server and handles the request. The main disadvantage of this approach is that data redistribution does not complete until each of the remapped records receives at least one request. The time it takes to complete a redistribution is therefore unbounded, which is a problem for servers that need to shut down quickly. For this reason, we did not consider this solution any further.

Another task where alternative strategies could have been chosen is the way to deal with requests while redistribution is in progress. In particular, if a hosting server receives a request for a record that is not yet shipped, the server simply handles the request locally since no other copy of the record exists. However, if the record has already been shipped, different options exist. One option is to reject the request and let the client keep trying until the redistribution is completed. However, this option does not conform with our transparency goal.

Another option is to always handle the request locally independent of whether it is a lookup or an update request. In the case of an update request for a record that is already relocated, the authoritative hosting server propagates the record's modified value to its new hosting server in order to keep the two copies of the record consistent. In this approach, a server can report completion of a redistribution to the configuration service only after it has finished shipping its records *and* made sure that the values of all modified records have been accepted by the new hosting servers. This solution has the advantage that update requests are processed slightly faster, but introduces additional complexity for keeping the records consistent.

A different approach can also be considered for the initial forwarding of requests. The initial server that is arbitrarily selected by a client to handle a request may forward the request directly to the new hosting server of the record involved instead of the currently authoritative one. If the record has already been shipped to its new hosting server, the request is handled immediately. If not, the new hosting server may either stall the request until the record is shipped to it, or it can fetch the record from its authoritative hosting server on demand. The former case does not satisfy the requirement of continuous availability. The latter case is a solution that we did consider, but whose advantages hardly outweigh the complexity it introduces.

There is a tradeoff between, on the one hand, forwarding a request to the record's authoritative hosting server and having it forwarded further if the record is already shipped, and on the other hand forwarding the request to the record's new hosting server and having the record fetched on demand if it has not been shipped yet. This tradeoff depends on the frequency and the types of requests that clients submit. The first strategy favors frequent lookups and rare updates as lookups are handled with no penalty, even for shipped records, when no updates occur. The second strategy favors more frequent updates as it eliminates the extra forwarding of every single request for a shipped record that has been updated. As it turns out, the first strategy is essentially simpler when also considering concurrency issues, which we discuss next.

5. The Solution for Overlapping Redistributions

In a reasonably sized distributed service consisting of a large number of servers, configuration changes requiring record redistributions may overlap. A realistic solution to our redistribution problem should therefore also operate in the case of concurrent configuration changes. In this section we show that our solution can easily be extended to support multiple, overlapping record redistributions.

The easiest way to deal with multiple concurrent redistributions is to simply apply a total ordering to them and execute them sequentially. This can be done by having the configuration service queue notifications for new configuration changes and process them one at a time. This solution is, however, not satisfactory since it does not achieve any concurrency. The redistributions are still handled one at a time.

In the following three approaches we attempt to introduce more efficiency by gradually introducing more concurrency for redistributions. In this section, let R_1, R_2, …, R_n be the sequence of upcoming redistributions and M_1, M_2, …, M_n their respective mappings. M_0 is the (current) authoritative mapping of the distributed service as a whole.

5.1. Approach I: Per-server Sequential Redistribution

A first step towards redistribution concurrency is to allow redistributions to overlap in the distributed service as whole but constrain each server to deal locally with just one redistribution at a time, completing redistributions in the order submitted. In this case, the configuration service does not need to queue notifications for new configuration changes. Instead, it generates a new mapping and distributes it to the servers as soon as it receives a notification for a new configuration change. The servers themselves are responsible for locally queuing incoming mappings and processing them one at a time in the order received.

Each server maintains a *queue of mappings*, which always contains *at least* one mapping. In the case of n redistributions in progress with mappings $M_1 \dots M_n$ and authoritative mapping M_0, a server's queue contains all these mappings in the order $M_0, M_1, \dots, M_n$. The mapping at the head of the queue is always the authoritative mapping as known by the server. The rest are mappings associated with the redistributions $R_1, R_2, \dots, R_n$ that are currently in progress.

A server that has relocated all records for redistribution R_1 can start carrying out the record relocation for the next redistribution R_2 before all other servers have completed redistribution R_1. However, the server does *not* remove either mapping M_0 or M_1 from its local queue of mappings. The authoritative mapping as known to the server (i.e., M_0) is removed from the server's queue only upon receiving a notification from the configuration service stating that redistribution R_1 has been completed by all servers. At this point, the server discards M_0 and replaces it by M_1, which becomes the new authoritative mapping.

To facilitate our description of this approach, as well as of the ones to follow, we define the *current redistribution* to be the oldest redistribution for which at least one server has not yet finished shipping records. Let R_1 be the current redistribution. Assume that a server has finished shipping records for $R_1, R_2, \dots, R_j (j \geq 1)$, and is now shipping records for R_{j+1}. During the shipping it receives a request for some record that was shipped based on $R_i (1 \leq i \leq j)$ and that thus cannot be handled locally. The server forwards the request based on the first mapping that remaps this record, which is mapping M_i. The server looks for such a mapping, starting at mapping M_1 and going no further than the mapping that is currently being handled by the server, that is, mapping M_{j+1}.

To make our description of the server's forwarding decision more precise, we introduce the notion of a virtual mapping. Consider a server S and a series of mappings $M_1 \dots M_n$. We define the *virtual mapping with first preference*

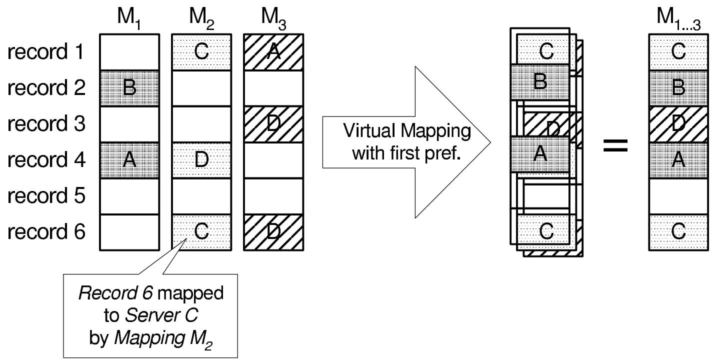
M1
M2
M3
record 1
record 2
record 3
record 4
record 5
record 6
Virtual Mapping with first pref.
M1...3
Record 6 mapped to Server C by Mapping M2

Figure 5.2: Virtual mapping with first preference

$M_{1...n}$ as the mapping that maps each record as prescribed by the *first* mapping in $M_1 \ldots M_n$ that maps it to another server than S, starting from M_1 and ending at M_n. The only records not remapped by the virtual mapping $M_{1...n}$ are the ones not remapped to another server than S by any of the mappings $M_1 \ldots M_n$. Figure 5.2 shows an example of six records being remapped by mappings M_1, M_2, and M_3 and the remapping of the same records based on the virtual mapping $M_{1...3}$.

Let us now explain how Approach I works. Upon receiving a notification for redistribution R_i, the configuration service builds a new mapping M_i and sends it to all servers. Mappings are delivered to all servers reliably and in the same order. When a server receives the new mapping M_i, it queues it if it is busy with some previous redistribution, or otherwise starts shipping records based on it. Only when a server has finished shipping all records based on a mapping, does it start shipping records based on the next mapping in its queue.

A record is shipped along with the index of the redistribution that mandated its relocation. The server receiving a record cannot reship it in the context of the same or any prior redistribution. This safeguards us against continuously shipping records back and forth between two or more servers. Such an anomaly could occur in the following scenario. Redistribution R_1 remaps a record that is initially in server A to server B and redistribution R_2 remaps it back to A. If A is working on R_1 while B is working on R_2, the record keeps being sent back and forth. Sending the index of the redistribution with the record prevents this situation.

```
ON Notification for Redistribution R[i] DO
  Compute mapping M[i] for R[i]
  Distribute it to all the servers

ON Completion of Redistribution R[i] DO
  Notify all servers about R[i]'s completion.
```

Figure 5.3: Configuration Service's pseudocode for Concurrent Approach I

A server notifies the configuration service when it completes shipping records for a redistribution. The server continues with processing redistributions until all the mappings in the server's queue have been processed. The authoritative mapping M_0 at the server is removed from the head of its queue only after the configuration service has announced that all servers have completed the current redistribution. After the removal, the next mapping in the queue, M_1, becomes the new authoritative mapping.

Upon receiving a request that cannot be handled locally, the server forwards it based on the virtual mapping with first preference of all mappings in its queue, say $M_{0 \dots n}$. It forwards the request to the appropriate server, along with the index k of the actual mapping M_k that prescribed this forwarding. If the receiving server needs to further forward the request, it will do so according to the virtual mapping $M_{k+1 \dots n}$. Assuming that the record exists, the server that has the requested record will eventually be reached and will process the request.

The pseudocode in Figures 5.3 and 5.4 shows the actions that the configuration service and the servers have to take to implement Concurrent Approach I.

5.2. Approach II: Per-server Mixed but Ordered Redistributions

A second step toward increased concurrency is to ease the requirements on when a server can start shipping records according to one of its queued mappings. The main idea is that there are cases where a server does not need to complete a redistribution to start working on the next one. Assume a server is currently going through its set of records, checking which ones are to be shipped based on redistribution R_i and it comes across a record that is not remapped by R_i. The server can then ship this record based on a successive redistribution $R_j (j > i)$, even if it has not finished R_i yet.

```
ON receiving mapping M[i] DO
  IF currently working on an earlier redistribution THEN
    put M[i] at the end of the mapping queue
  ELSE
    start shipping records based on M[i]

ON finishing shipping records for M[i] DO
  report completion of record relocations for M[i] to configuration service
  IF have not reached the end of the mapping queue THEN
    start shipping records based on M[i+1]
    //M[i] is not removed from the queue yet

ON receiving a request from a client DO
  IF the request can be handled locally THEN
    handle the request locally
  ELSE
    forward the request based on the virtual mapping with 1st pref M[k+1..j]
    //k is the index of the last redistribution that relocated the record
    //the server is currently shipping records based on M[j]

ON receiving notification about completion of Redistribution R[i] DO
  remove M[i-1] from the queue
  make M[i] the authoritative mapping
```

Figure 5.4: Configuration Service's pseudocode for Concurrent Approach I

The main difference with the previous approach is the time when records are shipped, not *which* records are shipped or *where* they are shipped to. In this approach the server ships each record as soon as possible, based on the virtual mapping with first preference of all the mappings in its queue. Requests are forwarded in the same way as in Approach I.

5.3. Approach III: Direct Shipping to Final Destination

Approach III deals with the forwarding inefficiency that arises when a record is shipped to different servers in a row. In both Approaches I and II, a record that is consecutively mapped to different servers by overlapping redistributions is shipped through each of them. The record finally ends up at the server mandated by the last redistribution.

The optimization introduced in Approach III entails that a record is shipped directly to the record's hosting server according to the *last* known redistribution. This policy keeps a record from being shipped from server to server when

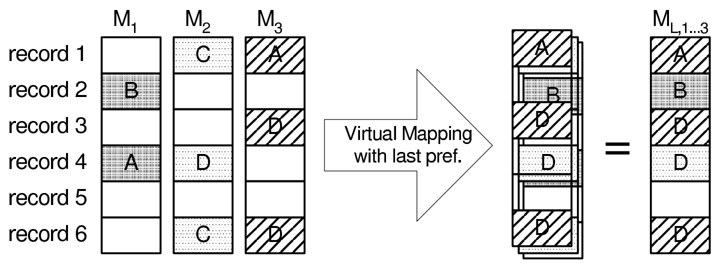

Figure 5.5: Virtual mapping with last preference

it is already known that it needs to be shipped further. Instead, the record is sent directly to the last server in the chain of servers it is mapped to. This policy prevents unnecessary network traffic and redistribution delay.

To explain, we need to introduce a second virtual mapping. Consider a server S and a series of mappings $M_1 \ldots M_n$. We define the *virtual mapping with last preference* $M_{L,1\ldots n}$ as the mapping that maps each record as prescribed by the *last* mapping that maps it to another server than S. Figure 5.5 shows an example of six records being remapped by mappings M_1, M_2, and M_3 and the remapping of the same records based on the virtual mapping $M_{L,1\ldots3}$.

Let us now see how Approach III works. As before, when the configuration service receives a notification for a configuration change, it generates a new mapping and distributes it to all servers. When a server receives a mapping, it places it at the end of its local queue of mappings. A mapping is removed from this queue only when the configuration service announces the completion of the respective redistribution. Since record redistributions are allowed to complete only in the order they were initiated, mappings are removed only from the head of a server's queue.

The main difference in Approach III is that a server ships records based on the virtual mapping with *last* preference of all the mappings in its queue. The records are thus directly relocated to the proper hosting server. However, servers still use the virtual mapping with *first* preference of all these mappings to forward requests that cannot be handled locally. This is done to avoid the following situation. Assume that M_1 is the last mapping in server S's queue, and server S ships a record to server A based on M_1. After having shipped the record, a new mapping M_2 arrives at server S remapping that same record to server B. If the virtual mapping with last preference was also used to forward requests, a request for this record would be sent to B, while the record may still be located at server A. Therefore, to ensure the request finds the record, it needs to travel through all servers that potentially store the record.

6. Related Work

In this paper we address the problem of data relocation in a distributed environment. A plethora of related publications have appeared in the literature, mainly dealing with relocating data in distributed databases, or in general, storage systems. However, the majority of these papers focus on different problems than the one we do. A number of them deal with the problem of determining the optimal *allocation* or *placement* of data in a set of devices or servers, usually trying to optimize load balancing and QoS characteristics [3, 4] or replication properties [5]. Unfortunately, they do not deal with the implications of the data transfer itself, or they assume a static data allocation that can be configured during a temporary (and probably partial) deactivation of the system [6]. Other papers deal with the details of how to carry out transactions while performing data transfers, but assume a model that supports replicated data [7].

In terms of providing a framework to add or remove servers, the problem we have tackled resembles that of dynamic changes to the set of servers in a distributed data storage system. When the set of servers changes, some data needs to be migrated to reflect the current set of servers. In many systems today such changes are made manually, by taking the system temporarily off-line. In other systems replication is employed to allow data to be redundantly stored in more than one server, to facilitate a smooth join or leave of a server. Many of the architectures that use replication in terms of adding or removing servers, are in fact dealing with fault tolerance, which is a problem orthogonal to the configuration problem we have presented.

Schemes like the ones described above do not apply to our situation, as we seek solutions to distribute data across servers without interrupting the service and without introducing replication. To the best of our knowledge, the problem described in this paper has not been addressed in the current literature.

7. Conclusions

This paper deals with a management issue of distributed services, namely the redistribution of non-replicated data among the servers comprising a distributed service. Our objective has been to redistribute the data without disrupting the service's availability. The main contribution of this paper is that we have shown that transparent data redistribution is possible. That is, it is possible to carry out such a redistribution in a way that is totally transparent to clients of the service. In order to exploit parallelism in the presence of overlapping configuration changes, we have also analyzed the implications of a concurrent version of the solution.

The solution consists of two parts. First, shipping the data records that need to be relocated to their new hosting server; second, updating the servers' mapping information to reflect the new configuration of the distributed service. Our solution enables low delays in the servicing of client requests during a configuration change, adds no significant processing requirement to the servers involved, and terminates in a timely fashion. Its most attractive advantage though is its conceptual simplicity, both in the sequential and the concurrent versions.

References

[1] H.-G. Hegering, S. Abeck, and B. Neumair. *Integrated Management of Networked Systems*. Morgan Kaufman, San Mateo, CA, 1999.

[2] E. Pitoura and G. Samaras "Locating Objects in Mobile Computing." IEEE Transactions on Kowledge and Data Engineering, 13(4):571-592, July 2001.

[3] E. Borowsky, R. Golding, A. Merchant, L. Schreier, E. Shriver, M. Spasojevic, and J. Wilkes. "Using attribute-managed storage to achieve QoS." In *Proc. of the 5th Int'l Conf. Workshop on Quality of Service*, Columbia University, New York, June 1997.

[4] J. Wolf. "The Placement Optimization Problem: a practical solution to the disk file assignment problem." In *Proc. of the ACM SIGMETRICS Int'l conference on Measurement and modeling of computer systems*, pp. 1-10, 1989

[5] A. Brunstrom, S. Leutenegger, and R. Simha. "Experimental Evaluation of Dynamic Data Allocation Strategies in a Distributed Database With Changing Workloads." In *Proc. Fourth Int'l Conf. on Information and Knowledge Management*, pp. 395–402, New York, NY, Nov. 1995. ACM, ACM Press.

[6] J. Hall, J. D. Hartline, A. R. Karlin, J. Saia, and J. Wilkes. "On Algorithms for Efficient Data Migration." In *Proc. 12th Symp. Discrete Algorithms*, pp. 620–629, New York, NY, Jan. 2001. ACM-SIAM, ACM Press.

[7] B. Kemme, A. Bartoli, and O. Babaouglu. "Online Reconfiguration in Replicated Databases Based on Group Communication." In *Proc. Int'l Conf. Dependable Systems and Networks*, Los Alamitos, CA, June 2001. IEEE Computer Society Press.

Authors addresses:
Spyros Voulgaris, Maarten van Steen, Aline Baggio, Gerco Ballintijn.
Vrije Universiteit Amsterdam
Faculty of Sciences
Department of Computer Science
De Boelelaan 1083a , 1081 HV Amsterdam
Amsterdam, The Netherlands
spyros@cs.vu.nl, steen@cs.vu.nl, baggio@cs.vu.nl, gerco@cs.vu.nl