Numéro d'identification:

Thèse

présentée à

L'Université Pierre et Marie Curie

en vue de l'obtention du titre de

Docteur de l'Université Pierre et Marie Curie

Spécialité :

Systèmes Informatiques

par

CORINA FERDEAN

Sujet de la thèse :

Enforcing Service-Specific Replica Consistency Models and Response Time Requirements for Heterogeneous Replicated Services

soutenue le 8 décembre 2006, devant le jury composé de :

Messieurs	Guy BERNARD	Rapporteurs
	Patrick VALDURIEZ	
	Florian Mircea BOIAN	Examinateurs
	Stéphane GANCARSKI	
	Mesaac MAKPANGOU (*Directeur de thèse*)	
	Pierre SENS	
	Marc SHAPIRO	

Abstract

Replication is largely accepted as a technique to improve the performance of popular Internet services, especially when they are accessed simultaneously by many wide-area dispersed clients. In the context of Replicated Service Hosting Systems, we address the problem of selecting the replica, that provides satisfactory response time, to the final clients and the problem of providing the appropriate consistency constraints for all replicas. Existing work recognizes the response time, as the metric that correlates directly with the user perceived performance. However, they don't take into account the real resource demands of the services and the current resource utilization. This is not appropriate when we consider that each service has its specific needs with respect to the system and the network resources that it uses. Also, the resources are heterogeneous, with respect to their capacities and load. We propose a response time estimation approach, where the response time is decomposed into independent components (service times and waiting times), and each component is approximated separately, by taking into account the capacity and the current utilization of the resources, needed by the service workload. The service supplier expresses the performance he/she wishes to be observed by the clients, by specifying a bound or by requiring the best possible response time. The response time requirement is enforced by a replica selection protocol, configured with the criterion, defining what is the suitable replica which should perform the clients' requests. We implemented this solution and the results, obtained experimentally, showed satisfactory fulfillment of the response time requirements, at a reasonable cost.

Existing work on replica consistency management offers a limited set of guarantees, for all the replicated services. However, different consistency constraints are pertinent from a service to another. We identify three consistency dimensions: quality of observable state, scheduling and dependency control. We propose a consistency meta-model, which reifies existing consistency guarantees, by identifying the parameters characterizing each dimension. The meta-model is attached a consistency protocol, which enforces service-specific consistency models, specified statically by the service suppliers. A service-specific consistency model contains the safety constraints, needed for the service to function correctly, when it is replicated, and the liveness constraints needed for bounding the discrepancy between the state of a particular replica and the ideal replica state. The protocol is built up of three independent building bricks, corresponding to the three orthogonal consistency dimensions. The main thesis message is that our replication approach was able to manage the heterogeneity of the services, with respect to their operation semantics and resource demands, when providing the appropriate replica consistency guarantees and performance requirements.

Résumé

Le problème que je traite dans la thèse est comment offrir de manière efficace les contraintes de cohérence et les requis de temps de réponse, demandés par des services repliqués hétérogènes, accédés à large échelle. La réplication est une technique classique utilisée pour améliorer la performance des applications Internet populaires, surtout quand elles sont accédées concurremment par un grand nombre de clients, dispersés à travers le monde. Les travaux existants ne prennent pas en compte la diversité des demandes des services pour différentes ressources système et réseau. Cette hypothèse relève une difficulté importante du problème de l'estimation du temps de réponse. Une autre difficulté est déterminée par la hétérogénéité des capacités des ressources et par la variabilité des disponibilités des ressources. Nous proposons une approche d'estimation, où le temps de réponse est décomposé dans des composants indépendants (temps de service et temps d'attente), et chaque composante est approximée séparément, en prenant en compte les capacités et les utilisations courantes des ressources demandées par le service. Le requis sur le temps de réponse est spécifié par le fournisseur du service, par un seuil qui correspond à la moyenne de temps de service des requêtes, ou comme critère d'optimisation, demandant le meilleur temps de réponse. Le requis est realisé par un protocole générique qui détermine quel est le réplicat approprié à choisir pour répondre aux requêtes de chaque client, parmi tous les réplicats existants. Nous avons implementé cette solution et les résultats expérimentaux obtenus ont montrés la satisfaction du requis sur le temps de réponse, à un coût raisonnable.

Les approches existantes pour la gestion de la cohérence des réplicats, offre un ensemble limité des garanties pour tous les services. Tout de même, la variété des sémantiques des services, demande des combinaisons des garanties différentes d'un service à l'autre. Nous proposons un meta-modèle de cohérence, qui réifie les garanties de cohérence existantes, en identifiant les paramètres qui caractérisent chaqu'une des trois dimensions de la cohérence: contrôle de divergence, contrôle de concurrence et contrôle de dépendances. Le meta-modèle est attaché un protocole de cohérence générique, qui réalise des modèles de cohérence spécifiques aux services. Un modèle de cohérence, spécifié par le fournisseur du service, contient des contraintes de sûreté, nécessaires pour que le service fonctionne correctement lorsqu'il est répliqué, et des contraintes de vivacité, nécessaires pour limiter la discrépance entre l'état d'un réplicat particulier et l'état du réplicat idéal. Le protocole de cohérence assemble trois briques de base indépendantes, qui correspondent aux trois dimensions orthogonales de la cohérence.

Le message de la thèse c'est que notre approche de réplication gère la hétérogénéité des services en terme de sémantiques des opérations et de demandes de ressources, en satisfaisant les contraintes appropriées concernant la cohérence des réplicats et la performance observé par les requetes des clients.

Acknowledgements

Firstly, I want to thank to my PhD supervisor Mesaac Makpangou for receiving me in his team and for teaching me many valuable principles about doing research during almost five years. I appreciate especially that he understood that my abilities in doing research evolve slowly, and that sometimes I need more time for a given task than scheduled. He never accepted an intuitive solution without asking me to motivate it with solid reasons. This has been a useful lesson, which helped me to become convincing not only when I speak computer science. I think I was lucky to have severe supervisor, because in this way I could accept the "no mercy reviews" for my papers submitted in the first two years of my PhD thesis. Not in the last place, I am grateful to him for understanding my extra-professional problems and for giving me advices about how to handle them.

I am very grateful to the project experts: Marc Shapiro, Pierre Sens et Bertil Folliot, for their advices that saved my life in conference presentations and for their ability to help me find my way in research when I was lost.

I thanks all my colleagues at Inria and LIP6 for being there professionally et humanely, always so "gentils". A special thanks to Ikram Chabbouh, Nguyen Thi Le Chau, Nicolas Gibelin, Ahmed Jebali, Fabrice Le Fessant, Simon Patarin, Nicolas Richer, Ahmed Mokhtar, Pierre Sutra, Jean-Michel Busca and to our secretary Thi Thanh van Tran. You are a great team!

I address a sincere "thank you" to the anonymous reviewers of my papers submitted to WIAPP'03, CIPC'03, DOA'04, Medianet'04, ICD-CIT'04, DOA'05, CDUR'05, SAINT'06. Their tecnhical remarks helped me improve the quality and the precision of my research work.

Contents

Part I

Introduction

Chapter 1

Introduction

1.1 Context

The target of our work is a *service*, which encapsulates data, kept in memory or stored on disk. It provides to clients a well-defined interface, consisting of operations that manipulate the data. In this interface, each operation is characterized by its signature and by its access type (read or update). Each service belongs to a service supplier.

The evolution of the service usage is characterized by a growing need for service suppliers to deliver their services to clients in a scalable manner. The suppliers expect that the service delivery respects some performance goals, in terms of the service availability, of the response time or the data throughput, that will be perceived by the clients. The performance goals must be fulfilled, despite a variable number and distribution of clients.

Under this usage context, a centralized approach, where a single server performs all the clients' requests, is unsuitable. The limitations of the centralized approach arise, for example, in the following two cases: a large number of requests received simultaneously by the central server, and large network distances between the server and the clients. In the former case, the server overload causes clients to suffer delays. This is the case, for example, of an e-commerce site offering a popular product, demanded by many clients spread through Internet. The latter case concerns the clients, localized far away from the server. These clients experience large delays, especially when they perform network-intensive requests, while the network is congested by concurrent traffic.

A traditional solution to the scalability issue is to replicate the service code and data (that the code manipulates) at several well-chosen hosts. This solution is currently employed in Replicated Service Hosting Systems [65], that represent the context of our work. The Replicated Service System relies on a hosting infrastructure, consisting of machines dispersed all over the world. Machines have various capacities, with respect to processor speed, disk I/O bandwidth, disk storage capacity, memory size and network connection quality. Service suppliers confide their services to a Replicated Service Hosting System, in order to make them accessible to clients, while guaranteeing good performance, in terms of response time that will be perceived by the clients. In order to fulfill the performance goal, the Replicated Service Hosting System replicate the service on several machines. A replicated service becomes a group of replica servers. A replica server (or simply *replica*) denotes a service instance, containing the data and the program, that the clients use in order to access the data. The number and the distribution of the clients accessing the replicas are unpredictable and vary dynamically. The Replicated Service Hosting System must also preserve the data correctness for each replicated service.

A Replicated Service Hosting System offers benefits both to the final clients and to the service suppliers. Replicating the service helps improve the clients' experience, with respect to the service availability, the response time perceived for their requests or the data throughput. The service remains accessible, even if some of the replicas become unreachable (e.g. because their network connection goes down). Response time is reduced, by selecting for the clients a proximal replica, a replica that is underloaded or a replica that is well-connected, in terms of network bandwidth. The benefits offered to the service suppliers include improved service throughput (in terms of the number of requests served per unit of time). By distributing client requests among the replicas and by reducing the traffic on WAN links, replication also helps reduce the host and network resources that are needed in order to serve the requests.

However, replicating services in a large-scale environment, raises a number of difficult issues, which need to be addressed, in order to improve client-perceived performance and replica consistency. We place these issues on two levels. The first level include the following issues:

- What is replicated? (only the service code or also its data?) and what is the granularity of replication? (are the data fully replicated or only partially?)
- What is the appropriate policy guiding the replica placement and how is it enforced? (on what hosts new replicas must be placed?) How is the minimum and the maximum number of replicas that could be created?
- What consistency constraints are needed, so as to preserve the service data correctness and the quality of replica state observed by the clients? How are these constraints enforced?
- What is the policy that defines the suitable replica that must be selected for each client? How is this policy enforced?

The second level contains the adaptability issues, which demand to take the right replica creation and replica selection decisions at run-time, according to the current status of the resources, which vary dynamically. This issues include:

- When a new replica is needed?
- When an existing replica cease to be useful?
- When another replica must be selected for a given client, because the current replica becomes inappropriate?

These are difficult issues, mainly because of the large-scale envisaged, where both replica hosts and clients are dispersed all over the world. Another source of difficulty is that the number, the distribution and the frequency of the clients requests vary unpredictably.

The issues, we are interested in, include replica selection and replica consistency management. Precisely, we enforce the criterion defining the suitable replica that should perform each clients' request and the service-specific criterion defining the required replica consistency.

1.2 Problem definition

In the context of Replicated Service Hosting Systems, the problem we address in this thesis is how to efficiently enforce the requirements, needed on response time and on replica consistency, by services, with various resource demands, while being accessed by many (concurrent) clients, spread all over the world.

This is a difficult problem to be addressed, because of the following reasons:

- The services are heterogenous. The operation types, the commutativity and the conflicting relationships, vary between operations of the same service. The quality of the replica state needed to be observed by a given operation is different from the needs of other operations of the service. The heterogeneity of services determine a large variety of consistency constraints, which could be required by different services.
- The resource demands vary from a service to another and for different operations offered by the same service. Some operations are CPU-intensive. Other operations are disk I/O-intensive or network bandwidth-intensive. The demands of other operations include arbitrary combinations of CPU, disk I/O or network bandwidth resources.
- The resources of the replica hosts have various processing capacities.
- The load of the resources varies dynamically.
- The service suppliers have different preferences with respect to the response time, that has to be provided to the clients.

Under these assumptions, the main challenge is to develop a Replicated Service Hosting System capable to accommodate efficiently the diversity of consistency constraints and performance requirements, needed by heterogenous services. The other challenges are particular to the replica consistency issue, respectively to the performance issue. The challenges, concerning the replica consistency issue, include: representing uniformly the possible consistency constraints and developing a consistency protocol, customized with service-specific models, enforced at a reasonable overhead. The challenges, concerning the performance issue, include: determining the impact of various system and network resources on the response time expected from the available replicas and determining how the metrics, that characterize the resources utilization, interfere with each other.

1.3 Summary of existing approaches

1.3.1 Summary of replica consistency approaches

By analyzing the state of art on the replica consistency management, we identified three consistency dimensions: quality of observable state, concurrency control and dependency control. With respect to the quality of observable state dimension, existing approaches (such as Tact [76], Refresco [45], PDBREP [1]) define the discrepancy between the current replica state and the ideal replica state, by bounding the total weights of unpropagated updates and the delay of updates propagation, at each replica. With respect to the scheduling of concurrent updates, existing approaches (such as Bayou [69], IceCube [30], Leganet [22]) take into account the ordering relationships between non-commutative accesses and the conflict resolution policies, to be applied for exclusive accesses. With respect to the dependency control, existing approaches (such as Bayou, IceCube) specify for each access, the set of previously accepted updates, on which it depends.

We identify two main limitations of existing approaches. The former limitation consists in accommodating the diversity of service semantics, requiring various hybrid models, which combine constraints available in existing models. Although existing systems already propose hybrid replica consistency models, there are some services for which neither of the existing consistency models is suitable. The services may require other combinations of constraints, such as Bayou's session guarantees and IceCube scheduling relationships or Tact numerical error/staleness and IceCube scheduling relationships. At the best of our knowledge, such combinations are not provided by existing systems. An example of a service, that needs the latter combination, is the digital library, such that a search on a given topic could not miss more than 10 articles and the modifications to the same article are conflicting. The latter limitation concerns the extensibility of existing approaches with new constraints, which could be pertinent for particular services. For example, as far as we know, existing approaches don't take into account the current replica state (e.g. the weights of the stabilized updates), when bounding the discrepancy between the state of a given replica and the ideal replica state. Also, they don't define asynchronous propagation conditions, based on the current replica state or on the transfer instant of related updates (i.e. so as to propagate several updates at the same time).

1.3.2 Summary of response time-driven replica selection approaches

By analyzing the state of art on response time-driven replica selection (including Carter et al. [7], RaDar [53], Web++ [74]), we identified several metrics defining the suitable replica, selected according to the expected response time. These metrics include system load, round-trip time, and/or network bandwidth. Each of these metrics is good for specific cases. For instance, the round-trip time metric is appropriate for Web documents of small sizes. The available bandwidth is relevant when the replies, returned to the clients, contain data of large size.

The main limitation of existing replica selection approaches is that they limit the environment conditions considered (i.e. they ignore the resource utilization). They also ignore the request characteristics, in terms of resource demands. Existing approaches usually configure statically the parameters that guide the decision that the system takes, in order to fulfill the objective of improving response time. However, in practice, the requests of different services (and even the requests of the same service) make use of arbitrary combinations of several resources, among CPU, disk I/O and network bandwidth. We argue that an accurate response time estimation approach should take into account the resource demands of the client requests and the processing capacity and the current utilization of those resources. Our approach takes into account these parameters, as we showed that they have a significant impact on the response time.

1.4 Summary of the solution

1.4.1 Support for specifying the consistency and performance requirements

The Replicated Service Hosting System doesn't know in advance the exact consistency constraints and the performance requirements, that are needed by all the services, simply because those demands vary from a service to another. That is why, we propose a customizable Replicated Service Hosting System that provides the right degree of replica consistency and the response time requirements, needed by each service. We assign to the service suppliers the responsibility to specify them, as he/she is the one who knows the needs of his/her service. Precisely, for each service that has to be replicated, its supplier provides the centralized version of the service (including the service logic and data), the service description, the policies needed for resolving conflicts and for ordering non-commutative operations and the discrepancy tolerated between the state of any replica and the ideal replica state and the requirements on the response time expected for each operation. The service description contains for each operation: the signature, the workload defining the demands for CPU, disk I/O and/or network bandwidth, the access type (read or write) and the impact of each operation on the data it modifies. The requirements on response are also specified per operation. Each requirement imposes a threshold or claims for the best possible response time. The threshold corresponds to the average response time of the clients' invocations to that operation. The Replicated Service Hosting System has the responsibility to implement the consistency management, enforcing the consistency criteria specified by the service supplier, and the replica selection criteria, that match the response time requirements, also specified by the service supplier. In this way, we minimize the effort of the service supplier. However, we assume that the service supplier specifies compatible consistency constraints and performance requirements. Such compatibilities concern mainly the type of concurrency control: pessimistic or optimistic. For example, stringent response time requirements claims for an optimistic concurrency control (because a pessimistic concurrency control could induce significant delays).

1.4.2 Replication model

We note, r_i, $i = 1,\ n$, the available replicas of a service. We consider that the replication degree n varies dynamically. The replicas are spread all over the world. We use the active replication model: a client initiates an *invocation* at any replica, called the *initiator*. If the operation is an update, then the *initiator* propagates the invocation to its *peers*, where it is re-executed. We define the *ideal replica* as the service instance, whose state is obtained by applying in the right order the non-conflicting updates issued at all the individual replicas. Conversely, the state of any individual replica is a view of the ideal replica state.

We use the active replication scheme for two main reasons. The former reason is to avoid that the primary becomes a bottleneck, when there many clients which access the replicated service concurrently. In such cases, the primary overload would conduct to long delays experienced by the clients. The latter reason is to allow replicas to be accessed in isolation. This makes our approach suitable also for dynamic environments, where replicas may disconnect from the system, as it is the case with the P2P systems. In the replicated databases, the scheme equivalent to active replication, is called *multi-master* or *update everywhere replication* [47].

We use update propagation in order handle updates uniformly for all services, independently of the underlying data type and content (which are information needed by a data propagation scheme).

Each client interacts with the replica server, that he/she has been assigned, within a session of requests. We use the term "request" as the synonym of "operation invocation". We use interchangeably the terms of "replica selection" for a client and "binding" of a client to a replica. However, there is a slight difference between the two terms, that regards the support needed, so that the client could interact with the replicated service. In this respect, the binding comprises the actions needed by the client so as to obtain the stubs and the selection of the initial replica with which the client will interact. The suitable replica may be re-selected during the client's session.

We define the abstraction of *host class* as a group of hosts, that provide similar response times when executing the same service, under 0 utilization (i.e. without contention). A host class may encapsulate the physical characteristics of the processor (e.g. speed, number), of the disk (e.g. storage capacity, I/O bandwidth), the software characteristics concerning the operating system (e.g. the scheduling policy) and the communication protocol stack. We assume a static decomposition of the hosting infrastructure into *domains*. A *domain* is a set of proximal hosting machines, where the distance is computed according to some metrics. Such metrics quantify, for example, the topological or the geographical proximity between the hosts.

1.4.3 Enforcing the service-specific consistency guarantees

In order to satisfy the service-specific replica consistency guarantees, we propose a flexible fine-grained replica consistency management approach, consisting of three building bricks: a consistency meta-model, a replica consistency protocol enforcing any service-specific consistency model and the replica wrappers. In order to define the meta-model, we abstracted existing replica consistency constraints, by identifying the parameters, characterizing each one of the three consistency dimensions: divergence control, scheduling and dependency control.

The meta-model is instantiated into a service-specific consistency model, by choosing appropriate parameters values, attached to sets of operations. The parameters values include the divergence metric, the functions for conflicts detection and resolution, the functions for ordering non-commutative operations, and the dependency types. The meta-model can be easily extended, in order to include other divergence metrics. We propose a new divergence metric, that takes into account also the accepted updates (besides the unpropagated updates), when quantifying the discrepancy between the state of a given replica and the ideal replica state. The service-specific consistency models are realized by a common replica consistency protocol, that relies on independent components, called *resolvers*. Each *resolver* enforces the constraints defined on a particular consistency dimension. The replica wrapper represents the replicated version of the service, by integrating the consistency management code, transparently for the final clients (i.e. they access the service using the same interface, as in the centralized case).

1.4.4 Enforcing the performance requirements

The main building brick of our approach for enforcing the performance requirements consists in a response time estimator, which provides the approximative response time expected by running requests with a given workload, on a given host, for which the availability of its resources varies dynamically. The response time estimator is parameterized by the workload of the request, and by the capacity and by the utilization of the resources that the service needs. The estimator decomposes the response time, observed for a given request, into independent components: the processing times, during which each resource is used by the request and the times, during which the request is waiting for each resource. We model the waiting time, using the regression-based statistical method. This method determines automatically the resources with the most significant impact on the response time observed for the request and the weights, denoting the degrees of their significance. In particular, the CPU waiting time becomes an exponential function, whose exponent depends mainly on the CPU capacity and on the CPU utilization. Using the measures of CPU waiting times obtained for some workloads, under some CPU utilization values, we were able to compute the CPU waiting time for any workload, under arbitrary CPU utilization value. This statement also remains true for the disk I/O waiting time and for the network transfer time. The client remains bound to his replica, until the percentage of requests, that violated the response time requirement, exceeds a given bound (e.g. 20%). When this bound is exceeded, the protocol selects another replica for the client.

The response time estimator represents the basis for our replica selection criterion, inferred automatically from the performance requirements, formulated by the service supplier. The replica selection criterion uses the response time estimator, when the resources are shared among several applications running concurrently. Otherwise, when the resources are underloaded, the selection criterion uses only the resource capacity, which is a static metric. The replica selection criterion, customizes a generic replica selection protocol, that we developed independently of particular response time requirements, of the service workload or of the current resources status. The protocol checks during the client session, if the current replica provides the needed response time. If a given percentage of the client requests doesn't verify the response time requirements, another replica is selected for the clients. The usage of domains helps performing scalable replica selection, with respect to the number and to the distribution of replicas.

1.4.5 Contributions

The contributions of our work are the following:

- A consistency meta-model, that aggregates abstract constraint types, attached to groups of operations; this meta-model can be instantiated into service-specific consistency models, containing arbitrary combinations of existing consistency guarantees.
- A replica consistency protocol, capable to enforce service-specific consistency models.
- A replica consistency framework, which integrates the service logic with the replica consistency management, transparently for the service suppliers and for the clients.
- A response time estimator, that takes into account the workload of the requests, so as to infer what are the relevant resources for each request to the service and the degree with which they impact the response time. The estimator exploits the processing capacity and the current utilization of those resources; we showed that our estimation algorithms can afford to use obsolete measures, as long as the discrepancy between those measures and the most recent ones doesn't exceed a given threshold (i.e. 20%).
- A request redirection protocol, that adapts the replica selection criterion and the rebinding criterion to the performance requirements. The redirection protocol exploits mainly the response time estimator.

The results obtained made the object of the papers published in the Proceedings of ICDCIT'04 [18], of DOA'05 [19], of CDUR'05 [20] and of SAINT'06 [21].

1.5 Outline of the document

This document is structured in four parts. The first part, entitled **Introduction**, contains one chapter with the same title. This chapter presents the context of our work, presents the problem that we address in this thesis and a summary of our solution.

The second part is entitled **Providing Service-Specific Replica Consistency Models** and contains three chapters. Chapter 2, entitled **Survey of Replica Consistency Management Systems**, describes systems providing replica consistency management. Chapter 3, entitled **Representing Service-Specific Consistency Models**, contains the definition and the object-oriented representation of the consistency meta-model, that we propose in order to express arbitrary combinations of guarantees attached to groups of operations. At the end of the chapter, we show how the meta-model is instantiated for each service, from an XML description of the service interface and of the consistency constraints, required by the service supplier. Chapter 4, entitled **A Generic and Customizable Replica Consistency Protocol**, shows how various service-specific consistency models are realized by the same generic protocol. In the beginning of the chapter, we depicts the building blocks composing the replica consistency framework. Then, we describe each of the building block in turn: the Contract object, the resolvers enforcing the constraints on the quality of observable state, on the scheduling and on the dependency control dimensions, the Consistency Manager and the Server-Side Replica Wrapper. We show how the three protocols are assembled into the consistency protocol, providing the replicated execution of each access. We prove that the protocol works correctly (i.e. it satisfies the liveness and the safety properties), by representing it by a distributed state machine, wherein the transitions are configured by the consistency constraints contained in the model.

The third part is entitled **Response-Driven Replica Selection** and contains four chapters. Chapter 5, entitled **Survey of Replica Selection Systems**, describes the replica selection strategies adopted by various systems, so as to improve the response time perceived by the clients. Chapter 6, entitled **A Workload-Aware Response Time Estimator**, presents an innovative approach for estimating the response time expected from the available replicas. This approach exploits the capacity and the utilization of the resources, that are relevant for the service's workload. Then we decompose the response time into independent components, including: CPU service time, disk I/O service time, CPU waiting time, disk I/O waiting time and network transfer time. We propose a formula, for computing each of these components. We present experimental validation of our approach, showing satisfactory estimation accuracy, especially for the workloads containing one bottleneck resource. Chapter 7, entitled **A Generic and Customizable Replica Selection Protocol**, presents our approach for enforcing the replica selection criterion, determined so as to satisfy the performance requirements. In the beginning of the chapter, we show the general specification of the performance requirement, including the characterization of the workload generated by the service. Then we abstract the replica selection and the rebinding criteria and we show how they are specialized so as to satisfy particular performance requirements. Then we describe the replica selection and the rebinding protocols, relying on domains as the support for scalable replica management. We present the protocol implementation and the experimental results, proving the benefits of rebinding the client dynamically to another replica, when the current one provides bad performance. Chapter 8, entitled **The Overall Replicated Service Hosting System**, begins by introducing the Information Repository component. Then, we present the distribution and the interaction between the system-specific components of the Replicated Service Hosting System, including: Information Repository, Response Time Estimator, Host Monitor and Metrology Server. We show how the framework integrates the service logic with the replica consistency management, transparently for the service suppliers and for the clients. This feature relies on a Server-Side Replica Wrapper, that is automatically generated for each service, from the XML specification of the service-specific consistency contract. Then, we present the distribution and the interaction between the service-specific components, including: Consistency Manager, Replica Manager, Server-Side Replica Wrapper and Client-Side Replica Wrapper. In the last section, we show the interface by which the Replicated Service Hosting System is made accessible to the service suppliers and to the clients.

The fourth part, entitled **Conclusion and Perspectives**, contains two chapters. Chapter 9 summarizes the replication approach that we propose, in order to enforce the consistency constraints and the performance requirements, that are needed by a given service. Chapter 10 depicts the key ideas for other complementary issues, which are not completely addressed in our work. The document ends with the enumeration of the perspectives that our work opens.

Part II

Providing Service-Specific Replica Consistency Models

Chapter 2

Survey of Replica Consistency Management Systems

2.1 Introduction

There are a lot of existing replication systems, that target various performance objectives (such as improved response time, service availability, servers load-balancing), while enforcing the suitable replica consistency constraints. The consistency constraints allow to obtain various consistency models, belonging to the range delimited by the two standard criteria: *strong consistency* and *eventual consistency*. The *strong consistency* guarantees that any read sees the result of the most recent write. The *eventual consistency* guarantees that all replica states become equivalent at some point in time, when all updates arrive at all replicas, and no more update is issued.

We surveyed the database and the service replication systems, which provide the standard *strong consistency* (*one-copy serializability*) correctness criterion and the systems which relax this criterion, in order to improve the performance of queries and of update transactions. We classify these systems in four classes: systems providing strong consistency, systems providing divergence control, systems performing relaxed concurrency control and systems providing hybrid models. The former class includes the atomic broadcast primitive used in Isis, or the Ricart & Agrawala's mutual exclusion protocol [57]. The second class include the systems like S2LDDC [52], Tact [76], PDBREP protocol [1], Refresco [45], Leganet [22] which control the quality of data observed by reads, by bounding the discrepancy tolerated between the local replica and the ideal replica. The third class is divided in two subclasses, containing the systems, which perform pessimistic, respectively, optimistic scheduling. The former subclass include systems, such as Refresco [45] and Leganet [22]), which relax the serializability, by considering the commutativity between updates accessing different tuples. The second subclass include systems such as Bayou [69], IceCube [30], optimistic atomic broadcast [17], which avoid the overhead induced by the total order provided pessimistically, by allowing replicas to be accessed independently, and reconciliating a-posteriori the updates which are conflicting or wrongly ordered. The fourth class include systems like Khazana, which combines the features of the former classes.

This rest of this chapter is structured in five sections. Sections 2.2, 2.3, 2.4 and 2.5 present, respectively, the systems providing strong consistency, the systems providing divergence control, the systems performing relaxed concurrency control and the systems providing hybrid models. Section 2.6 concludes the chapter.

2.2 Systems providing strong consistency

Strong consistency is enforced by the atomic broadcast primitive defined by Birman [60] or by the Ricart & Agrawala's mutual exclusion protocol.

The atomic broadcast primitive (ABCAST), defined by Birman in 1987 [60], and implemented in group communication systems (e.g. Isis, Horus) guarantees that all accesses are delivered in the same order at all replicas. Each replica maintains locally a logical clock. When an access is issued, its initiator replica decides the access ordering by obtaining the largest logical clock, within a two-phase commit protocol.

The Ricart & Agrawala's mutual exclusion protocol allows each access to proceed, only after receiving the permissions from all the replicas, participating to the scheduling of the accesses. This protocol guarantees that for any two conflicting updates only one is executed. It also guarantees that every access sees all the accesses issued before it (and which aren't conflicting with it).

One copy serializability The equivalent of strong consistency for replicated databases is *one-copy serializability* (used in Ingres [47]). This criterion states that the effects of update transactions submitted independently at different replicas are the same as if all the updates had been applied on a single copy [47]. It contains two sub-properties: mutual consistency (requiring that all replicas have the same values) and serializability.

Both mono-master and multi-master database replication systems enforces one copy serializability by using the eager replication technique, where the refresh transactions are propagated to secondaries within the boundaries of the initial transaction [47]. The acceptance or the rejection of the transaction is decided by a two-phase commit protocol.

The strong consistency criterion isn't appropriate in a large scale environment, with potentially many accesses issued concurrently and many replicas. In such a context, reads may experience significant delays, because they must wait for all previously issued updates to be applied locally. This is the reason why various existing systems relaxed the strong consistency. In this respect, they allow accesses to be applied concurrently at various replicas and/or they bound the discrepancy between the local replica state and the ideal replica state.

2.3 Systems providing divergence control

This category of systems control the quality of replica state observed by reads and updates. In this respect, they characterize the quality of the replica state by the difference between the local replica state and the ideal replica state. The right quality of the replica state is obtained by bounding this difference.

The consistency protocol called *strict 2-phase locking distributed divergence control (S2LDDC)* [52] enforces the *epsilon-serializability* criterion. This criterion imposes bounds on two divergence metrics: *imported inconsistency*, associated to queries, and *exported inconsistency*, associated to update transactions. The *imported inconsistency* defines the total amount of modifications not observed by the query, although these modifications have been integrated on other replicas. The *exported inconsistency* quantifies the effects of the update transaction on the data that it modifies. The protocol works by decomposing the imported/exported inconsistency into local conditions, which bound the amount of inconsistency read from/introduced to the local database maintained by each replica.

Tact [76] proposes an innovative approach for bounding the divergence between the state of a particular replica and the state of the ideal replica, that would have been obtained if all issued updates have been correctly executed. The divergence between a particular replica state and the ideal replica state is expressed using the three metrics: *staleness*, *numerical error* and *order error*. The *numerical error* counts the total weights of updates applied at peers, but not seen by the local replica. The *staleness* of a replica is defined as the difference between the current time and the issuance time of the oldest update unseen locally. The *order error* metric is defined as the number of tentative updates applied at the local replica and not yet scheduled. They satisfy the divergence metrics bounds, by decomposing them into local conditions, which become the invariants maintained by each replica (by pushing the local updates to the peers).

Refresco (Routing Enhancer through FREShness COntrol) [45] proposes a fine-grained flexible replica consistency model, which is instantiated into specific models, imposing thresholds on the metrics: *age* (equivalent to staleness), *order* (equivalent to numerical error when the weights are 1) and *card*. The *card* metric counts for the number of stale tuples. The freshness requirements are associated to the database, to relations or to attributes. A finer granularity increases the concurrency among non-conflicting transactions.

A significant contribution of the systems Refresco [45] and Leganet [22] is to minimize the number of refresh transactions to be applied before the query, so that the freshness requirements are satisfied. In this respect, they compute a sequence wherein they include successively the remaining refresh transactions (committed at the master, but not yet propagated locally). Each time a refresh transaction is included in the sequence, the values of the divergence metrics are updated. This iteration ends when thresholds, imposed on the divergence metrics, are satisfied.

The PDBREP protocol [1] provides fine-grained freshness management of queries, under configurations, where each database may be partitioned over several sites. In this context, the distributed queries access objects (i.e. relations or partitions of relations) located at various sites. Each object has associated a master replica, to which all updates are redirected. The objects represent the granules for freshness requirements. Precisely, the users may specify the freshness requirements, in terms of the version numbers of the objects that their queries access. Before a query starts, all objects, that it accesses, are brought to the same version, which is at least the version number required by the query. A version is maintained during the query execution by locking the accessed objects, by using freshness locks. Their approach combines the two propagation modes: pushing updates from the master at the secondaries, and pulling refresh update transactions at the secondaries from the master. The former mode is used when the secondary nodes are idle (i.e. there is no on-going query or refresh transaction running at that nodes). The latter mode occurs when the requested object doesn't respect the freshness level required by the query.

Pacitti et al. [43, 44] measure the freshness as the ratio between the number of refresh transactions committed at the secondaries and the number of transactions committed at the master. Their consistency policy also bounds the percentage of tuples accessed concurrently by refresh transactions and queries. They distinguish between the propagation strategy where each refresh transaction is encapsulated within a single message (*deferred-immediate* strategy) and the strategy where each operation of a refresh transaction is encapsulated within a different message (*immediate-immediate* strategy). In each case, each message related to a refresh transaction is executed as soon as it arrives at secondaries. They proved experimentally that the immediate-immediate strategy improves the freshness of secondary copies (even when the network delay augments or when the available bandwidth decreases). This gain is due to the concurrency between the propagation of updates from the master and the delivery of refresh updates at the secondaries. Pacitti et al. [44] introduce another propagation strategy, called *immediate-wait*, where each operation of a refresh transaction is carried in a separate message, but the refresh transaction is executed after all the messages of the same refresh transaction are received. They showed, that this strategy improves freshness compared to *immediate-immediate*, when the updates arrive at the master at a high rate.

2.4 Systems performing relaxed concurrency control

This category of systems relax the total ordering, by taking into account the commutativity and the conflicting relationships between updates. We distinguish between systems performing pessimistic concurrency control and systems performing optimistic concurrency control. They perform the concurrency control before the access execution (preventing conflicts or wrong ordering), respectively, after the access execution (reconciliating a-posteriori conflicts or wrong ordering). The following two subsections detail the two categories of systems.

2.4.1 Systems performing pessimistic scheduling

Pacitti et al. [42] and Leganet enforce the total order only among the conflicting transactions. Two transactions are conflicting if the intersection, of the sets of tuples which they access, isn't empty. They rely the conflict detection policy on the transactions parameters.

Pacitti et al. [42] provide global FIFO ordering by an asynchronous approach, called preventive replication, where the master associates to each transaction a chronologically increasing timestamp. At a secondary, the execution of each refresh transaction is delayed by the time period corresponding to the maximum multicast delay. This delay can be eliminated if the FIFO guarantee is already provided by the underlying network. In this way, the secondaries apply the transactions in the same chronological order, in which they have been applied at the masters.

Leganet determines the sets of relations read and modified by a transaction, by parsing the corresponding transaction code.

2.4.2 Systems performing optimistic scheduling

Bayou [69, 70] is an optimistic replication system, that provide to users, monotonically increasing versions of the data that they access. They introduce the concept of *session guarantee*, with the following options: *Read Your Writes*, *Monotonic Reads*, *Monotonic Writes* and *Writes Follow Reads*. *Read Your Writes* and *Monotonic Writes* guarantees that the user sees all his previously issued updates, when he initiates a read, respectively an update. The *Monotonic Reads* and *Writes Follow Read* guarantees that the user sees the updates previously observed by his last read, respectively his last update.

Bayou controls the resolution of conflicting invocations, by means of two user-defined procedures, *detect* and *merge*, plugged into the system. The function *detect* returns true if the two accesses are conflicting or non-commutative. If this is the case, the function *resolve* is used, in order to determine what access to exclude (in the case of conflicts) or what access should precede the other (in the case of non-commutativity). In contrast to Bayou, in our approach both functions are generated from the service supplier's specification, which exploits the attributes and/or the arguments of the two accesses.

The optimistic replication protocol [17] extends the Atomic Broadcast to the optimistic replication scenario, in order to improve the response time of updates. Their optimistic assumption is that the sequencer, that determines the total order of updates, doesn't fail. If this isn't the case, the newly selected sequencer may determine an ordering which is different from the ordering already seen by the client. The algorithm solves such external inconsistencies client-side, by selecting the ordering already applied at a majority of replicas.

IceCube [30] is a generic scheduling framework, wherein the reconciliation of tentatively executed updates is customized by a large variety of ordering, dependency and exclusiveness relationships relating actions, that are initiated at the same replica or at different replicas. The reconciliation approach of IceCube is centralized and relies on grouping actions related by some constraints into clusters. The actions in a particular cluster are ordered independently from the actions in another cluster in iterative steps. At each step, the action with the highest weight is selected from the cluster and introduced in the schedule, and all actions conflicting with the selected one are deleted from the cluster. Also the parcel constraints are respected by introducing the corresponding actions in the cluster. The resulted schedule is applied at all replicas.

The applications built on IceCube [40, 51] take into account the arguments given to the operation calls, when defining the conflict resolution policies. Examples of such applications are the cooperative editor and the shared agenda. In the former application, a conflict occurs, for example, between two actions, creating a directory and a file, with the same name. In the latter application, a conflict occurs between two actions, which try to reserve the same room, at the same date and time.

In the context of multi-master replication, Martins et al. [35] propose a decentralized reconciliation algorithm, which extends the IceCube approach to dynamic large-scale systems, like P2Ps. This algorithm uses a set of reconciliers, each of them performing the reconciliation algorithm of IceCube, wherein the conflict resolution relies on application semantics. The resulted schedule is performed at each replica. An innovation of their work is to store the updates submitted by the clients at various replicas in the underlying DHT, maintained by the P2P system. In this way, if a replica disconnects, its updates are transferred automatically to its neighbors, by the P2P data management system. Another innovation of their work is to determine the number and the set of reconciliers needed, so as to minimize the reconciliation response time. They use a polynomial regression method, wherein the coefficients take into account the number of actions, the number of replicas, the bandwidth capacity and the number of nodes.

Daudjee et al. [14] introduce a new correctness criterion, called *strong session snapshot isolation*. It guarantees the strong snapshot isolation at the level of client sessions (defined as the sequence of transactions issued by a given client). Within a given client session, each transaction sees all previously committed transactions. Transactions which belong to different sessions may be executed in any order at different replicas. They proved experimentally that strong session snapshot isolation improves significantly the response time of queries and the transaction throughput.

DECAF (Distributed Extensible Collaborative Application Framework) [66] is a framework based on Model View Controller paradigm and which aims to improve the responsiveness of collaborative editors. This framework contains generic collaborative objects, whose replicas are updated optimistically. They express the trade-off between consistency and responsiveness by means of optimistic vs. pessimistic views. The former integrate all updates, while the latter

system	quality of observable state	concurrency control	dependency control
quasi-copies [2]	arithmetic condition, delay condition	-	-
Lazy replication [34]	-	-	client specified order
Tsae [25]	version number	total order, causal order	-
Bayou [69]	-	total order, check() and merge()	RYW, MR, MW, WFR
S-DSO [75]	service-specific semantic functions	-	-
Globe [29]	-	FIFO order	RYW, MR, MW, WFR
IceCube [30]	- -	best order mutual exclusiveness, alternative	parcel predecessorSuccessor
Cascade [10]		total order, causal order	RYW, MR, MW, WFR
Tact [76]	numerical error, staleness, order error	total order	-
Fluid replication [12]	periodical propagation	optimistic/pessimistic mode last writer policy	-
Khazana [67]	time-bounded option modification option, consistency option	total order, causal order availability option, rejection policy	-
Refresco [45]	age, order, card	FIFO order	-
Leganet [22]	card	total order	-

Table 2.1: Options provided by existing replica consistency systems

only the committed updates. Each update is executed optimistically at the replica where it has been submitted, where the optimism regards three types of assumptions: the objects read by the update are stable (they integrate only committed transactions), no committed update has been missed locally, no update is modifying the same objects concurrently. If the primary copy confirms the three assumptions, the corresponding update is committed, otherwise it is aborted.

2.5 Systems providing hybrid models

This category of systems combine the features of the systems providing divergence control and the features of the systems performing relaxed concurrency control.

Khazana [67] is a flexible replica consistency management system, that combines several types of constraints. The constraints on the quality of observable state, define the time when remote updates must be made visible locally, the time when local modifications should be visible at peers and the degree of tolerated staleness. The constraints on the scheduling define the execution mode of accesses (i.e. pessimistic vs. optimistic) and the rejection policy, in the case of conflicting updates.

2.6 Conclusion

Existing replica consistency management approaches enforce constraints on three dimensions. We name them *quality of observable state*, *concurrency control* and *dependency control*. In the following three subsections, we characterize these dimensions and enumerate the constraints provided by existing systems on each dimension (as summarized in Table 2.1).

2.6.1 The quality of observable state dimension

The *quality of observable state* dimension limits the discrepancy between the current replica state (observed by the accesses performed on the replica), and the ideal replica state. Transient discrepancies are quantified by means of divergence metrics, that exploit various operation attributes. We classify existing divergence metrics in four classes: *delay metrics*, *staleness metrics*, *weight metrics* and *order metrics*. The former two classes exploit the issuance time of accesses, the third class exploits the weights of accesses and the latter class exploits the number of accesses executed tentatively. These metrics are associated threshold values, that must be satisfied before an access could proceed.

Delay divergence metrics The oldest and widest means to characterize the divergence between a current replica and the ideal replica relies on the propagation delay of updates. The propagation delay metric defines the maximal time period that elapses between the update's issuance and its propagation to the peers. It is used in quasi-copies [2], under the name *delay condition* particularized for each update, in Khazana [67], under the name *time-bounded consistency option*, in Fluid replication [39], where the propagation is performed periodically. In S-DSO [75], the propagation delay of updates is computed by using user-supplied semantic functions.

Staleness divergence metrics A widely-used condition on the quality of observable state, limits the time period during which each replica may remain stale, with respect to its peers. The staleness divergence metric is used within: Tact [76], timed-consistency [71], epsilon-serializability [52], delta consistency in Beehive [63], consistency dimension in Khazana, staleness threshold in [33], as the *age* metric in Refresco [45].

Weight divergence metrics The weights, (as defined in Tact and in Sabbarus [28]), quantify the severity of read operations with respect to the quality of the local data that they will observe, respectively the impact of update operations on the data that they will modify. Weight-based divergence metrics include: the *arithmetic condition* used in quasi-copies, the *imported inconsistency* and the *exported inconsistency*, both used in *epsilon-serializability* [4], the *numerical error* used in Tact and in Sabbarus, the *modification bound* option used in Khazana, the *version number* used in Tsae [25], in PDBREP protocol [1] and in Refresco, under the name of *order*, the *card* metric used in Refresco [45] and in Leganet [22].

The *arithmetic condition* was defined in the case of numerical data. It bounds the arithmetical difference between the value of the local replica and the value of the ideal replica.

The *version number* counts the number of updates issued at peers and that are unseen locally yet. In this case, the weight of any update has the value 1.

A weight-based divergence metric is assigned a numerical threshold, that must be satisfied by the state of a given replica, before an access could proceed.

Order-based divergence metrics The order-based divergence metrics improve the correctness of the replica transient state, with respect to the number of tolerated conflicts and to the number of updates wrongly ordered. They include: the *order error* used in Tact and the *bounded ignorance* [32]. Bounded ignorance relies on the abstraction of a *N-ignorant transaction*. This is a transaction that could proceed even if it has not seen yet the effects of at most N transactions, that should precede it in the serialization order.

2.6.2 The concurrency control dimension

The goal of concurrency control is to ensure the correctness of the replicated state and to apply the same set of updates, in the right order, at all replicas (i.e. non-commutative accesses are totally ordered). The scheduling of an access decides its execution order within the global history, with respect to concurrent accesses that do not commute or that are exclusive with it. The service supplier adapts the concurrency control, by means of scheduling relations defined between pairs of operations. A scheduling relation, applied to two accesses, defines what is their execution order, if the two accesses aren't commutative or what access to eliminate, if the two accesses conflict.

Ordering of non-commutative updates We distinguish between traditional ordering types and service-specific ordering relationships. With respect to the traditional consistency models, in this case, the ordering is applied only to non-commutative updates. The traditional ordering types include: total ordering, causal ordering, FIFO ordering, server-specified order and global order. The total ordering is used in the sequential consistency model [37], in the linearizability model [78], in Bayou [70], in Cascade [10], in Tsae, in Tact, in Khazana, in multi-master database replication scheme, as showed in [42].

Lamport's causal ordering and FIFO ordering relax total ordering. The causal ordering considers implicitly that operations, that are not causally related, are commutative. It is used in the causal consistency model [37], in Cascade, in Tsae, in Khazana. FIFO ordering considers that all operations are commutative, and it is intended merely to preserve the user-intended action sequence. It is used in the FIFO consistency model [37] and in Globe [29].

The global FIFO order is used in the mono-master database replication scheme [42], so as to enforce the commitment of refresh transactions in the order of their associated transactions at the masters.

Server-specified order and *global order* are used in [34]. *Server-specified order* requires that all invocations to a given operation are totally ordered. *Global order* requires that all invocations to all operations are totally ordered.

Service-specific ordering relationships define a partial ordering among all accesses based on user-specified operation semantics, such as commutativity or a favorite order. The commutativity of operations is explored in the conflict matrix model [3], in the generic broadcast [46], in IceCube [30]. The favorite order is explored in IceCube by means of a constraint, called *best order*. It is defined between pairs of actions issued at the same replica or at different replicas. IceCube also defines a constraint, called *predecessorSuccessor*, that relates two actions, issued at the same replica, so that the latter should be applied only if the former executed successfully.

Conflict resolutions Correct update scheduling also includes a mechanism for dealing with conflicts, if they can appear. Dealing with conflicts involves two types of constraints. The former specifies if potentially conflicting accesses could execute simultaneously. It makes the distinction between optimistic and pessimistic accesses. Examples of systems supporting this constraint type are: Khazana, within the *availability consistency dimension*, and Fluid replication [12]. The latter contains options for detecting and resolving conflicts. Bayou proposes user-specified procedures for detecting if two invocations are conflicting, called *dependency check procedures*. Conflicts resolution is based on *client-provided custom procedures* in Coda [31], Ficus [56] on *merge procedures* in Bayou, *last writer policy* in WebFS [73] and Fluid replication, update rejection policy in Khazana, an object scheduling constraint called *mutually exclusiveness* in IceCube. In the case of rejected actions, IceCube also proposes a constraint, called *alternative* relationship, that makes a non-deterministic choice between a given set of actions.

Snapshot isolation *Snapshot isolation* was introduced by Berenson et al. [4], so as to improve the transaction response time by relaxing serializability. It distinguishes between queries and updates. The queries read the latest snapshot of the database. The update transactions accessing different data items (i.e. non-conflicting transactions) are allowed to proceed concurrently (while the conflicting updates are totally ordered). Snapshot isolation is provided in Oracle, Microsoft SQL Server, InterBase, PostreSQL [4].

2.6.3 The dependency control dimension

Providing dependency-based guarantees is necessary for the correct execution of operations, that depend on already stabilized operations. If an access depends on a set of updates that have been already executed at peers, then those updates must be integrated into the local copy, before the access.

The dependency-based guarantees include the Bayou's session guarantees used in [69], in Globe, in GlobeCBC [58] and in Cascade, the client-specified order used in [34] and the *parcel* relation defined in IceCube. The user uses the client-specified order, in order to specify dynamically during the service execution, the dependency between the current read and previously issued updates, that must be applied before the read. The *parcel* relation requires that all actions within a given set are executed.

With respect to the state of art, we envisage that there are services that may need other hybrid models, obtained by combining existing consistency constraints and possibly new consistency constraints (that could be inserted dynamically in our framework). Such hybrid models must be enforced efficiently, even if the availability of the resources varies dynamically, in a large-scale environment.

Chapter 3

Representing Service-Specific Consistency Models

3.1 Introduction

Different services require different consistency models (each model containing a set of constraints). The issue that we address in this chapter is how to represent uniformly all existing replica consistency constraints into a single unified meta-model. We argue that fine-grained consistency management is needed (e.g. at the level of a set of operations), because different constraints types suit different operations and different execution environments. For example, the Bayou's Read Your Writes option is pertinent for reads and the propagation delay for updates. Also, only the optimistic concurrency control mode can be used when the network conditions vary unpredictably. In order to accommodate various service needs and execution environment characteristics, we abstract the existing consistency constraints on each of the three orthogonal consistency dimensions: *quality of observable state*, *concurrency control* and *dependency control*. We encapsulate the abstract constraint types, associated to sets of operations, into a meta-model. The meta-model can be instantiated for each service by the service supplier with a minimal effort, by choosing the appropriate constraints. The appropriate consistency model is specified using XML.

The rest of this chapter is structured in three sections. Section 3.2 presents the consistency meta-model. Section 3.3 describes the new divergence metric that we introduce. Finally, section 3.4 concludes the chapter.

3.2 The consistency meta-model

Definition A consistency model consists of a set of constraints. Each consistency constraint addresses one of the three independent issues that any consistency management system is concerned with. These issues are: what is the quality of the replica state that each invocation needs to observe, what are the scheduling guarantees that have to be provided for non-commutative or conflicting concurrent invocations, what are the dependencies to be preserved for new invocations with respect to the updates that have been already applied at

consistency dimension=quality of observable state | concurrency control | dependency control
quality of observable state constraint=(divergence metric, predicate())
divergence metric=predefined metric | (user-defined metric, conit())
predefined metric="numerical error" | "order error" | "staleness" | "propagation delay" | "stabilization delay" | "relative divergence"

concurrency control constraint=concurrency control mode | conflict semantics
conflict semantics = (scheduling relation type, detect(a_1, a_2), resolve(a_1, a_2))
concurrency control mode ="pessimistic" | "optimistic"

scheduling relation type = "conflicting" | "non-commutativity"
dependency control constraint=predefined dependency | (user-defined dependency, predicate(a_1, a_2))
predefined dependency=RYW | MW | MR | WFR | Lamport happens-before

Figure 3.1: The formalization of existing consistency constraints

all replicas. We consider three consistency dimensions, corresponding to these issues. We call them: *quality of observable state*, *concurrency control* and *dependency control*. A constraint on the *quality of observable state* dimension enforces the progression of replicas towards an equivalent state. One enforces such a constraint, by spreading the locally issued updates to the peers, and by bringing the relevant remote updates from the peers to the current replica. A constraint on the *concurrency control* dimension enforces replicas convergence, by deciding the acceptance/rejection and the execution order of conflicting or non-commutative updates, which have been executed tentatively. A constraint on the *dependency control* dimension determines the set of updates, that have been already stabilized, and that must precede a given access at all the replicas, because the access must see the impact of those updates on the local data.

We define a consistency meta-model, in order to represent existing and possibly new consistency models. The meta-model enables each service supplier to instantiate the consistency model that offers the suitable consistency guarantees to its clients, according to the semantics of the replicated service. In order to define the meta-model, we identify the parameters that characterize the constraints on each consistency dimension, together with the options available on these parameters.

3.2.1 Abstracting the consistency constraints

3.2.1.1 Constraints on the quality of observable state dimension

A constraint on the *quality of observable state* dimension bounds the discrepancy tolerated between the current replica state and the ideal replica state, by means of two parameters: a *divergence metric* and a *predicate*, stating what values of that metric are valid. The *divergence metric* parameter is specialized into one of the following two parameters: *predefined metric* and *user-defined metric*.

The options, available for the parameter *predefined metric*, include the metrics defined in TACT [76]: *numerical error*, *order error*, *staleness* (defined in section 2.6) and the metrics *propagation delay*, *stabilization delay* and *relative divergence*. The predicate on the *predefined metric* usually requires that the metric values should be inferior to a given bound. The predicate on order error isn't a sufficient condition for performing the reconciliation. Precisely, if the number of updates performed tentatively at each replica is small, they never get scheduled, if only order error is used. That is why, we introduced the metric called *stabilization delay*. This metric defines the maximum time period during which an update may remain tentative, before being reconciliated with its concurrent updates (which have also been applied tentatively).

The service suppliers use the *user-defined metric* parameter, in order to define a new divergence metric, by means of a function that provides the *conit* value (associating a numerical value to the current replica state). The value of the *conit* is updated when one of the following events occur: reconciliation of tentative updates, propagation of updates to peers or reception of updates from peers. The computation of the *conit* exploits the access weights or the timestamps of the previously mentioned events.

3.2.1.2 Constraints on the concurrency control dimension

A constraint on the *concurrency control* dimension has two parameters: *concurrency control mode* and *conflict semantics*. The *concurrency control mode* parameter specifies if the concurrent execution of potentially conflicting or non-commutative updates is admitted or not. It provides two alternative options: *pessimistic* and *optimistic*. The former option defers the execution of updates, until their acceptance (together with their execution order) or rejection is decided. With the latter option, the updates are applied tentatively before being scheduled.

The *conflict semantics* parameter specifies how to handle conflicts and non-commutativity. It contains the scheduling relation type (conflicting or non-commutativity) and the Bayou's functions *detect* and *resolve*, parameterized by a pair of accesses.

3.2.1.3 Constraints on the dependency control dimension

A constraint on the *dependency control* dimension specifies the causal dependencies between new updates and updates, that have already been stabilized. It contains one of the following two parameters: *predefined dependency* and *user-defined dependency*.

The options available for the *predefined dependency* parameter include the session guarantees of Bayou [69]: *RYW* (Read Your Writes), *MW* (Monotonic Writes), *WFR* (Writes Follow Reads), *MR* (Monotonic Reads) and *Lamport happens-before* [78].

The *user-defined dependency* parameter specifies service-specific dependencies, by means of a predicate, that takes as arguments two accesses and returns true if the former access depends on the latter.

```
<!ELEMENT service (interface, consistency_constraints)>
<!ATTLIST service name CDATA>
<!ELEMENT interface (operation+)>
<!ATTLIST interface name CDATA>
<!ATTLIST operation id CDATA>
<!ATTLIST operation signature CDATA>
<!ATTLIST operation weight CDATA>
<!ATTLIST operation type CDATA>
<!ELEMENT consistency_constraints(quality_of_observable_state*, concurrency_control*,
dependency_control*)>
<!ATTLIST quality_of_observable_state operations CDATA >
<!ATTLIST quality_of_observable_state numerical_error CDATA>
<!ATTLIST quality_of_observable_state order_error CDATA>
<!ATTLIST quality_of_observable_state propagation_delay CDATA>
<!ATTLIST quality_of_observable_state relative_divergence CDATA>
<!ATTLIST quality_of_observable_state user-defined_divergence CDATA>
<!ATTLIST concurrency_control type CDATA>
<!ATTLIST concurrency_control stabilization_delay CDATA>
<!ATTLIST concurrency_control pair_operations CDATA>
<!ATTLIST concurrency_control condition CDATA>
<!ATTLIST concurrency_control resolution CDATA>
<!ATTLIST dependency_control dependency_type CDATA>
<!ATTLIST dependency_control pair_operations CDATA>
<!ATTLIST dependency_control user_dependency CDATA>
```

Figure 3.2: The DTD of a contract specification

Figure 3.1 summarizes the existing constraints on each of the three consistency dimensions.

3.2.2 Fine-grained consistency

Different consistency constraints may be relevant for different sets of operations. For example, RYW concerns the invocations issued by the same caller. Propagation delay is defined for updates. We argue that fine-grained consistency management, at the operation level, is needed, in order to provide to each access, the mandatory constraints that it needs. In this respect, the consistency meta-model associates each consistency parameter to an operation or to a pair of operations.

3.2.3 Specifying a service-specific consistency contract

A particularity of our approach is that the functions contained in the constraints exploit the arguments passed to invocations. This feature claims for the service interface to be known. We call *service-specific consistency contract*, the specification of the service supplier, that includes both the service interface and the consistency constraints. We use XML, as the language support for specifying a service-specific consistency contract.

3.2.3.1 A DTD for service-specific consistency contracts

Figure 3.2 shows the DTD for service-specific consistency contracts, specified in XML. It contains the interface declaration and the definition of the consistency constraints. An operation is specified using the tag *operation*, that has four attributes: operation identifier, operation signature, weight and type (read or update). The weight is an arithmetic expression, that uses the operation arguments.

A consistency constraint is represented by one of the following three tags: *quality_ of_ observable_ state*, *concurrency_ control* and *dependency_ control*, corresponding to the previously identified dimensions. The *quality_ of_ observable_ state* tag has one attribute for each predefined divergence metric and the attribute *user-defined metric*, that extends the system with new metrics. In the case of a predefined divergence metric, the attribute value contains the bound imposed on the metric. It is defined by a number, in the case of *numerical_ error* and *order_ error*, and by a time period (e.g. "15 min"), in the case of *staleness*, *propagation_ delay* and *stabilization_ delay*. If the attribute used is *propagation_ delay*, the user should also specify within the attribute *operations*, the identifiers of the operations, to which the constraint is attached. The value of the attribute *user-defined metric* contains the name of the class, that encapsulates the logic for computing the conit associated to the replica and the predicate imposed on the conit, so as to bound the divergence between the local replica state and the ideal replica state. This class is provided by the service supplier in a separate source file.

The *concurrency_ control* tag specify the concurrency control mode (optimistic or pessimistic), that is common to all the operations. It has the attribute *stabilization_ delay*, that defines how long each access could wait for its acceptance decision.

The *concurrency_ control* tag has four attributes: *pair_ operations*, *type*, *condition* and *resolution*. These attributes are used for specifying the policies needed for conflicts detection and resolution. The *pair_ operations* attribute contains the identifiers of the two operations, to which one policy is attached. The *type* attribute contains the scheduling relation: conflicting or non-commutativity, that could occur between pairs of accesses to those operations. The *condition* attribute contains the boolean expression for detecting conflicts or non-commutativity, between a pair of accesses. The *resolution* attribute contains the expression for deciding what access to exclude, if the two accesses are conflicting or what access is the predecessor, if the two accesses are not commutative. The expressions for *condition* and *resolution* define implicitly the pair of access groups, to which the resolution policy applies.

The *dependency_ control* tag has the *dependency_ type* attribute, for specifying one or several predefined dependencies, and the *user_ dependency* attribute, containing the service-specific condition for two accesses to be causally related. This condition is attached to a pair of operations, whose identifiers are specified in the *pair_ operations* attribute.

3.2.3.2 Examples of consistency contracts

As an example, we consider an e-learning application, containing two services that provide access, respectively, to two types of data: courses and articles. The courses are manipulated by five operations: addCourse, modifyCourse, deleteCourse, readCourse and searchCourse. The articles are manipulated by five op-

```
<service name=”Courses”>
 <interface name=”ICourses”>
  <operation id=”addCourse” signature=”short addCourse(char* name, char* text)”
type=”update”/>
  <operation id=”modifyCourse” signature=”short modifyCourse(char* name, char* text)”
type=”update”/>
  <operation id=”deleteCourse” signature=”short deleteCourse(char* name)” type=”update”/>
  <operation id=”readCourse” signature=”char* readCourse(char* name)” type=”read”/>
  <operation id=”searchCourse” signature=”char* searchCourse(char* topic)” type=”read”/>
 </interface>
 <consistency_constraints>
  <quality_of_observable_state operations=”addCourse modifyCourse deleteCourse”
 propagation_delay=”0 sec”/>
  <concurrency_control type=”pessimistic”/>
  <concurrency_control pair_operations=”(modifyCourse, modifyCourse) (modifyCourse, delete-
Course) (deleteCourse, deleteCourse)” type=”non-commutativity” condi-
tion=”id_op1.name==id_op2.name” resolution=
“id_op1”/>
  <concurrency_control pair_operations=”(addCourse, addCourse)”
type=”conflicting” condition=”id_op1.name==id_op2.name” resolution=”(id_op1.weight ≥
id_op2.weight) ? id_op1 : id_op2”/>
 </consistency_constraints>
</service>

<service name=”Articles”>
 <interface name=”IArticles”>
  <operation id=”addArticle” signature=”short addArticle(char* name, char* url)” type=”update”/>
  <operation id=”modifyArticle” signature=”short modifyArticle(char* name, char* url)”
type=”update”/>
  <operation id=”deleteArticle” signature=”short deleteArticle(char* name)” type=”update”/>
  <operation id=”readArticle” signature=”char* readArticle(char* name)” type=”read”/>
  <operation id=”searchArticle” signature=”char* searchArticle(char* topic)” type=”read”/>
 </interface>
 <consistency_constraints>
  <quality_of_observable_state numerical_error=”100” order_error=”10” staleness=”1 day”/>
 <concurrency_control type=”optimistic”/>
 <concurrency_control pair_operations=”(modifyArticle, modifyArticle) (modifyArticle, deleteArti-
cle) (deleteArticle, deleteArticle)” type=”non-commutativity” condi-
tion=”id_op1.name==id_op2.name” resolution=”id_op1”/>
 <concurrency_control pair_operations=”(addArticle, addArticle)” type=”conflicting” condi-
tion=”id_op1.name==id_op2.name” resolution=”(id_op1.weight ≥ id_op2.weight) ? id_op1 :
id_op2”/>
 <dependency_control dependency_type=”RYW”/>
 </consistency_constraints>
</service>
```

Figure 3.3: Examples of consistency contracts for e-learning services

erations: addArticle, modifyArticle, deleteArticle, readArticle and searchArticle.

Figure 3.3 shows two examples of consistency contracts, specified in XML, for the two services. The contracts respect the following semantics: the students see the same list of courses, with the same content, at any replica they access. However, the list of available articles or their content may diverge from a replica to another. Consequently, the operations manipulating courses have a pessimistic concurrency control mode, while the operations manipulating articles have an optimistic concurrency control mode. We impose that local updates to courses should become visible immediately at all the peers, while in the case of local updates to articles, we admit a propagation delay of “1 day”.

Two modifications or two deletions to the same course or to the same article should be totally ordered. Two insertions of a course with the same name, or of an article with the same name, are mutually exclusive. If a conflict occurs, then the item accepted is the one with the biggest weight. We limit the number of articles that are missed, when searching for articles on a given topic (the missed articles haven't been transmitted yet to the current replica). We also impose a RYW constraint on searchArticle, so that an author will see his own articles when he/she performs searches.

3.2.4 Validity of option combinations

Not all combinations of options make sense. For example, Lamport happens-before can't be used when the concurrency control mode is optimistic. Precisely, in the optimistic scenario, commutative updates can't be causally ordered with respect to updates applied locally tentatively. This happens because commutative updates are accepted immediately and any accepted update is applied before all tentative updates. Also, the order error divergence metric can be used, only when the concurrency control mode is optimistic. In our work, we assign to the service supplier the responsibility of specifying valid combinations of options.

3.3 The relative divergence metric

A drawback of the existing divergence metrics is that they ignore the updates that are already in the stable history of a given replica. For example, if we consider the bank account service, individual replicas can afford a discrepancy of 100$ if the current account contains 10000$. However, 100$ isn't a tolerable discrepancy if the current account is 500$. We introduce a divergence metric, called *relative divergence*, that takes into account the current state of a given replica. A predicate on this metric bounds the range of values accepted for the local replica state, by means of an inferior and a superior threshold, defined relative to the ideal replica state.

We approximate the state of a replica by its *conit* (notion introduced in TACT [76], in order to assign a numerical value to the current replica state). We note $conit(r_i)$, the conit associated to the replica r_i. We note $conit(r)$, the conit associated to the ideal replica r. The guarantee on the *relative divergence*, to be satisfied by $conit(r_i)$, is expressed as follows:

$(1-p)*conit(r) \leq conit(r_i) \leq (1+p)*conit(r)$, where $p \in [0,\,1]$ is a system-defined parameter.
This guarantee is equivalent with:
$-p \leq (conit(r_i) - conit(r))/conit(r) \leq p$
$\Longleftrightarrow |conit(r_i) - conit(r)|/conit(r) \leq p, \forall\ i = 1,\ n$

An issue is the decomposition of the above guarantee, into local conditions, to be satisfied at each replica. Therefore, we define *conit* as the sum of the weights of the local unpropagated updates, the weights of the local propagated updates and the weights of the updates received from the peers. We associate to each replica r_i, $i = 1,\ n$, the following three sets: UU^i, PU^i and RU^i, where UU^i contains the updates issued at r_i and that are still unpropagated, PU^i contains the updates issued at r_i, already propagated and RU^i contains the updates received at r_i from its peers. We obtain the following formula for the *conit* associated to the replica r_i:
$conit(r_i) = \sum_{u \in UU^i} u.weight + \sum_{u \in PU^i} u.weight + \sum_{u \in RU^i} u.weight$, where $u.weight$ is the weight of the update u

Theorem: We consider n replicas r_i and r the ideal replica. If each replica enforces that: $(\sum_{u \in UU^i} u.weight)/(\sum_{u \in PU^i} u.weight) \leq p/(1-p)$ (a)
then $|conit(r_j) - conit(r)|/conit(r) \leq p, \forall\ j = 1,\ n$ (b)

Proof: This theorem shows the invariant which must be maintained by each replica, so that to realize, at any moment, the global bound on the relative divergence metric.

We associate the following variables, associated to replica r_i:

$lpstate^i = \sum_{u \in PU^i} u.weight$
$lustate^i = \sum_{u \in UU^i} u.weight$
$rstate^i = \sum_{u \in RU^i} u.weight$
$lstate^i = lpstate^i + lustate^i$
$conit(r_i) = lstate^i + rstate^i$

We define $conit(r) = \sum_{i=1}^{n} lstate^i$

From the assumption that $\forall\ u \in PU^i \quad \Rightarrow u \in RU^j$, where $j \neq i$ and
$\forall\ v \in RU^j \Rightarrow \exists\ is.t.v \in PU^i$
we obtain $\sum_{i=1, i\neq j}^{n} lpstate^i = rstate^j$

With these elements, the condition (a) becomes:
$lustate^i/lpstate^i \leq p/(1-p)$
$\Longleftrightarrow (1-p)*lustate^i \leq p*lpstate_i$
$\Longleftrightarrow p*lustate^i + (1-p)*lustate^i \leq p*lustate^i + p*lpstate^i$
$\Longleftrightarrow lustate^i \leq p*lstate^i$

We evaluate the condition (b) for the replica r_j
$\Rightarrow \sum_{i=1, i\neq j}^{n} lustate^i \leq p*\sum_{i=1, i\neq j}^{n} lstate^i \leq p*\sum_{i=1}^{n} lstate^i$

$\Rightarrow \sum_{i=1, i \neq j}^{n} lustate^i \leq p * \sum_{i=1}^{n} lstate^i$

But, $\sum_{i=1, i \neq j}^{n} lustate^i = conit(r) - \sum_{i=1, i \neq j}^{n} lpstate^i - lstate^j = conit(r) - (rstate^j + lstate^j) = \; conit(r) - conit(r_j)$
$\Rightarrow conit(r) - conit(r_j) \leq p * conit(r)$
$\Rightarrow (conit(r) - conit(r_j))/conit(r) \leq p$

But $conit(r) \geq conit(r_j)$
$\Rightarrow |conit(r_j) - conit(r)|/conit(r) \leq p$

Exploiting the *relative divergence* metric provides two main benefits with respect to the approach of TACT. Firstly, it takes into account the current state of the local replica and the state of the ideal replica, besides the total weights of the unpropagated updates. Secondly, the local conditions (a), inferred from the global guarantee (b), don't depend on the number of replicas. In practice, we need to bound the total weights of the propagated updates. One solution to this issue is to limit the size of PU^i, by considering only the updates propagated within a given time period.

Inferring the constraints from the weights Choosing the right consistency options and assembling them correctly within a consistency model is difficult. In order to reduce the burden on the service-suppliers, we envisage lighter specifications. We keep the granularity at the operation-level, distinguishing between reads and writes. The service suppliers assign per-operation weights. The weights are defined statically or computed dynamically by means of a user-defined function.

We translate the weight of a read, into the right degree of relative divergence that it can tolerate, and the weight of an update, into the transfer instant when it has to become visible at all the peers. We consider that the operation weights have the values range between 1 and max_weight, where max_weight is a system-defined parameter. We associate with the read a one constraint on the *relative divergence* metric, where the parameter p has the value $1 - a.weight/max\_weight$. In particular, if the weight of the read a has the maximum value, then a requires to see an up-to-date replica. We associate with the update a one constraint on the *propagation_ delay* metric. This constraint imposes that a must be propagated to the peers, before the time period given by $(1 - a.weight/max\_weight) * max\_delay$, where max_delay is the maximum propagation delay, pre-configured within the system.

3.4 Conclusion

We proposed a consistency meta-model, which aggregates existing and possibly new consistency constraints, by means of parameters attached to each of the three dimensions: *quality of observable state*, *concurrency control* and *dependency control*. A significant contribution of our work is the extensibility of the meta-model, which allow the service suppliers to introduce new divergence metrics (if they are needed). Another relevant result is the definition of a new divergence metric that takes into account also the stable state (i.e. the reconciliated updates) besides the unpropagated updates, when quantifying the discrepancy between the current replica and the ideal one.

Chapter 4

A Generic and Customizable Replica Consistency Protocol

4.1 Introduction

The issue that we address in this chapter is how to enforce any consistency model, that could be instantiated from the meta-model, by a single protocol. In this respect, we follow the decomposition of the consistency aspect among the three dimensions, that we have identified in the previous chapter: quality of observable state, concurrency control and dependency control. The constraints on each consistency dimension are resolved separately by a generic protocol, that they customize. The overall consistency protocol is composed of the protocol enforcing the quality of the observable state, the protocol enforcing the scheduling of concurrent invocations and the protocol enforcing the dependencies. The consistency protocol is encapsulated within a primitive, called for each access, transparently for the final clients, so as to provide the replicated execution of that access at all the replicas. We consider the case where the replicas are fully connected. We assume that the communication is reliable (i.e. the messages are received in the right order and no message is lost).

The rest of this chapter is structured in seven sections. Section 4.2 depicts the building blocks of the replica consistency framework, together with the functionality that they provide. Section 4.3 shows the reification of the abstract constraints by the Contract object. Sections 4.4, 4.5 and 4.6 show the algorithms needed for resolving the constraints, respectively, on the three consistency dimensions. Section 4.7 assembles the building blocks into the overall consistency protocol. Finally, section 4.8 concludes the chapter.

4.2 The consistency building blocks overview

The replica consistency framework contains the following building blocks: the Contract object, three resolvers attached to the consistency dimensions, the Consistency Manager and the Server-Side Replica Wrapper (as shown in Figure 4.1). This architecture has been originally proposed in BOAR system [5].

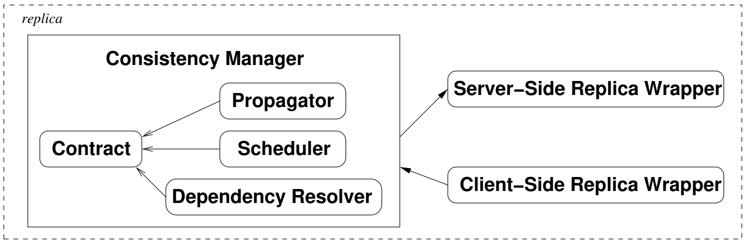

Figure 4.1: The replica consistency framework

The Contract encapsulates the objects representing the constraints contained in the service-specific consistency model.

We call consistency constraint resolver the entity responsible for enforcing the constraints on a particular consistency dimension. We distinguish between three different consistency constraint resolvers: Propagator, Scheduler and Dependency Resolver. The Propagator enforces the constraints on the *quality of observable state* dimension. It queries the StateMonitor contained in the Contract, in order to determine when the propagation of local updates should be performed. The Scheduler enforces the constraints on the concurrency control dimension, scheduling concurrent accesses. It is specialized into a PessimisticScheduler and a Reconciliator, according to the concurrency control mode: pessimistic, respectively, optimistic. The Reconciliator queries the StateMonitor in order to determine when the reconciliation of the tentative updates is needed. The Dependency Resolver enforces the precedence among causally dependent accesses. The three resolvers use five common data (Figure 4.5): the Server-Side Replica Wrapper, the identifier of the current replica, the Contract object, the LocalHistory object (containing the identifiers of all accesses applied on the current replica) and the list *incoming* of the updates received from peers, for which at least one predecessor is missing from the LocalHistory. The execution of the updates in *incoming* is deferred, until all their predecessors are applied.

The Consistency Manager enforces the consistency constraints required by the service supplier and which are encapsulated within the Contract object.

The Server-Side Replica Wrapper provides the interaction between the service and the Consistency Manager, transparently for the service suppliers and for the clients. In this respect, the Server-Side Replica Wrapper encapsulates the local service instance, on which it applies the accesses received from the peers. When a Consistency Manager is instantiated, it is configured with a Contract object, with the Server-Side Replica Wrapper and three resolvers: a Propagator, a Scheduler and a DependencyResolver.

The following five sections present each building block in turn.

4.3 The Consistency Contract

4.3.1 The representation of an access

We abstract an operation invocation, in order to apply the replica consistency protocol uniformly, for all the operations of any service. In this respect, we use

```
class Param { /* ... */ };
template class<ParamType>
class ParamWrapper<ParamType> : public Param {
  ParamType arg;
};
template class<ParamType_b>
class ArrayWrapper<ParamType_b> : public Param {
  short nb;
  ParamType_b* arg;
};

class AccessId {OperationId op, int call_count};
typedef vector<AccessId> AccessIds;
typedef map<ReplicaId, AccessIds> Predecessors;

class Invocation {
  vector<Param*> arguments;
  Param* result;
  AccessId id;
  CallerId caller;
};

class Access : public Invocation {
  int tmst;
  ReplicaId initiator;
  int weight;
  Predecessors preds;
};
```

Figure 4.2: Access representation

two C++ classes: Invocation and Access (Figure 4.2). An Invocation encapsulates: the actual arguments of the called operation, the result of the invocation, the invocation identifier and the caller identifier.

An object Access enriches an Invocation with the attributes needed so as to compute the conit. These attributes include: the timestamp of the invocation issuance, the identifier of the initiator replica, the weight, the list of the predecessors (i.e. the updates on which the current access depends), classified by their replica of issuance.

4.3.2 The representation of a service-specific consistency contract

We reify the consistency constraints in order to make them exploitable by the replica consistency management system. The reified constraints are encapsulated within a Contract object. This object provides the constraints needed for an access or for a pair of accesses. Figure 4.3 shows the C++ classes used for the representation of a service-specific consistency contract.

Reification of the replica state The LocalState object contains the identifiers of all accesses applied on the current replica, the locally issued updates not yet propagated to peers, the timestamp of the most recent update propagation and the timestamps of the most recent update receptions from the peers. The identifiers of the accesses already executed, are encapsulated within the LocalHistory object. Within this object, the accesses are classified by their replica of

```
typedef vector<OperationId> OperationIds;
enum ReplicationEvent {access_issuance, access_propagation, access_reception, access_scheduling,
access_scheduled};
typedef vector<Access> GroupAccesses;
typedef map<OperationId, AccessIds> PeerHistory
class LocalState {
  LocalHistory* history;
  QueuedAccesses* outgoing;
  int tmst_send;
  map<ReplicaId, int> tmst_rcved;
};
class LocalHistory {
  map<ReplicaId, PeerHistory> history;
  int getCallCount(OperationId& id);
};

class StateMonitor {
  LocalState* replica_state;
  TimePeriod stabilization_delay_bound;

 public:
  virtual void notify(ReplicationEvent event, Access& a) {}
  virtual bool isObservable(Access& a) { return true; }
  virtual bool isReconciliationNeeded() { return false; }

  virtual int getConit();
  int getStabilizationDelay() { return stabilization_delay_bound; }
  TimePeriod* getPropagationDelay(Access& a) { return NULL; }
};

enum dependency_type {RYW, MW, MR, WFR, Lamport_happens-before};
class PairAccesses {
  enum RelationType { non-commutativity, conflicting, none };
  Access* a1, * a2;
};

class PairOperations {OperationId op1, op2;};
typedef PairAccesses* (*ResolutionFct)(Access&, Access&)
typedef int (*WeightFct)(Access&)
typedef bool (*DependencyFct)(Access&, Access&)

class Contract {
  OperationIds updates_ids;
  map<OperationId, WeightFct> weights;
  StateMonitor* state_quality;
  enum ConcMode {pessimistic, optimistic} conc_mode;
  map<PairOperations, ResolutionFct> relations;
  vector<dependency_type> pred_dependencies;
  map<PairOperations, DependencyFct> user_dependencies;

 public:
  PairAccesses* detectAndResolve(Access& a1, Access& a2) { if (fct = relations[(a1.id.op, a2.id.op)])
return (*fct)(a1, a2); else return NULL; }
  bool isUpdate(OperationId& op) { return (op ∈ updates_ids); }
  bool needsScheduling(Access& a);
  int getWeight(Access& a) { return (*weights[a.id])(a); }
  TimePeriod* getPropagationDelay(Access& a);
};
```

Figure 4.3: Service-specific consistency contract representation

issuance. The accesses issued at a given replica are also classified by the target operation.

Reification of a constraint on a *divergence metric* The StateMonitor object contains the bound on the stabilization delay (defining how long each access could wait for its scheduling decision) and a pointer to the LocalState object. The StateMonitor defines the divergence metric by means of the *conit* (i.e. the numerical measure of the replica state). The computation of the *conit*, provided by the method getConit, relies on the LocalState object, and possibly on other service-specific data. These data are updated by the method notify, when one of the following events occur: the issuance of an access, the propagation of an update, the reception of an update, the initiation of an access scheduling or the decision of accepting/rejecting an access.

The predicate on the divergence metric is checked by the boolean method isObservable. This method takes as argument an access and checks the predicate on the current value of the *conit*. If the predicate isn't satisfied, this indicates that local updates should be pushed to peers or that the reconciliation of tentative updates is needed.

Representing constraints on predefined divergence metrics In order to represent constraints on existing divergence metrics, we specialize the StateMonitor into two subclasses: TactConditionMonitor and PropagationDelayMonitor (Figure 4.4). Both classes provide the method getPropagationDelay, that returns the bound on the propagation delay, associated to the access given as argument. A TactConditionMonitor object encapsulates bounds on the divergence metrics of TACT: numerical error, order error and staleness. In this case, the *conit* is computed as the total weights of the updates in *outgoing*. The method notify handles the event of an access issuance, by adding the weight of the access to the *conit* value and by incrementing the number of the local accesses in the tentative state. The propagation of local updates to the peers is needed, if the current value of the *conit* exceeds the local bound on the numerical error. The reconciliation is needed, if the number of the local tentative updates exceeds the bound on the order error.

A PropagationDelayMonitor object associates the maximum propagation delay to each operation of type update. The propagation delay is defined statically as a time period or it is computed dynamically, by using a function, automatically generated from the user-supplied specification of mappings (condition, time period).

The Contract object We define the type ResolutionFct, representing the functions used for the scheduling of concurrent accesses, that are potentially conflicting or non-commutative. A resolution function is invoked for two accesses, and it works as follows. If the accesses are non-conflicting and commutative, the function returns NULL. Otherwise, the function returns as result a PairAccesses object, containing the scheduling relationship between the two accesses (i.e. conflicting or non-commutativity) and how they are scheduled. If the two

```
class TactConditionMonitor : public StateMonitor {
  int numerical_error_bound, order_error_bound; TimePeriod staleness_bound;
  int conit, nb_local_tentative;
 public:
  TimePeriod* getPropagationDelay(Access& a) { return new TimePeriod(staleness_bound); }
  void notify(ReplicationEvent event, Access& a) {
    switch (event) {
      case access_issuance: conit+= a.weight;
       if (a.state == PENDING) nb_local_tentative++; break;
      case access_scheduled: nb_local_tentative - -; break;
    }
  }

  int getConit() { return conit; }
  bool isObservable(Access& a) { return (conit ≤ numerical_error_bound); }

  bool isReconciliationNeeded() { return (order_error_bound > nb_local_tentative); }
};

typedef TimePeriod* (*TransferInstantFct_sync)(Access&);
class TransferInstant {
  TimePeriod* period;
  TransferInstantFct_sync fct;
};
class PropagationDelayMonitor : public StateMonitor {
  map<OperationId, TransferInstant> propag_delay;
 public:
  TimePeriod* getPropagationDelay(Access& a);
};
```

Figure 4.4: Representing the predefined divergence metrics

accesses are related by a conflicting relationship, we consider two resolution policies. In the first policy, the PairAccesses object doesn't contain any access and the choice, of the access to reject, is left to the system. In the second policy, the PairAccesses object contains the access to be accepted (while the other is implicitly rejected) or the access that should replace the two conflicting accesses. If the two accesses are related by a non-commutativity relationship, the PairAccesses object contains the access that should precede the other, in the case of a user-favorite order. It doesn't contain any access, if the choice of the predecessor is left to the system.

With these elements, we define the Contract object (Figure 4.3) as a container containing the identifiers of the update operations, the mappings of weight functions to operation identifiers, a StateMonitor object, the concurrency control mode (optimistic or pessimistic), the mappings of resolution functions to pairs of operations, the set of predefined dependencies (i.e. RYW, MW, MR, WFR, Lamport_happens-before) and the mappings of user-dependency predicates to pairs of operations.

4.4 The propagation protocol

The Propagator guarantees the right degree of divergence between the local replica state and the ideal replica state, as defined by the propagation delays associated to operations and/or by the predicate encapsulated within the State-Monitor (primitive makeObservable in Figure 4.5). This predicate is checked at

```
ServerSideReplicaWrapper* wrapper;
ReplicaId crt_replica;
LocalHistory history;
Contract* contract;
GroupAccesses incoming;

class QueuedAccess { Access* a; TimePeriod delay; };
typedef vector<QueuedAccess> QueuedAccesses;
class Propagator {
  QueuedAccesses outgoing;
};

void makeObservable(Access& a) {
  state_quality->notify(access_issuance, a)
  if (!state_quality->isObservable(a))
    send(outgoing)
}

void fctPropagationThread(Access& a) {
  TimePeriod* propag_delay = state_quality->getPropagationDelay(a)
  if (*propag_delay == 0) {
    A = {a} ∪ {e.a/e ∈ outgoing, e.a ∈ a.preds}
    send(A)
  } else {
    get position i, such that Σ_{j=1}^{i} e_j.delay < *propag_delay ≤ Σ_{j=1}^{i+1} e_j.delay
    e ={a, *propag_delay - Σ_{j=1}^{i} e_j.delay}; e_{i+1}.delay = Σ_{j=1}^{i+1} e_j.delay - *propag_delay
    insert e in outgoing between e_i and e_{i+1}
  }
}

void propagate() {
  while (outgoing != ∅) {
    sleep(e_1.delay)
    A = {e_1.a}
    i=2; while (e_i.delay == O) {A = A ∪ {e_i.a}; i++}
    A = A ∪ {e.a/e ∈ outgoing, e.a ∈ a'.preds, ∀ a' ∈ A}
    send(A)
  }
}

void onReceiveUpdates(GroupAccesses& A) {
  for each a ∈ A {
    if (a ∈ history) continue;
    if (a.preds ⊆ history) {
      wrapper->execute(a)
      history = history ∪ {(a.initiator, a.id)}
    } else incoming = incoming ∪ {a}
  }
  execute ready updates from incoming
}
```

Figure 4.5: The Propagator

the time of an access issuance. If it evaluates to true (e.g. the bounds on the conit values are exceeded), then the Propagator pushes to the peers the unpropagated updates.

Figure 4.5 shows the propagation protocol run by Propagator, for any update (primitive propagate). The protocol exploits the list *outgoing* of the locally issued updates, not yet propagated. In this list, the updates are ranked ascendingly by their propagation delays. The propagation delay of an update in *outgoing* is defined relatively to the delays of the updates that precede it. Precisely, the propagation delay of an update is given by the sum of the propagation delays of the updates ranked before it and its own propagation delay.

The Propagator runs a thread, in order to send to the peers the updates in *outgoing* when their propagation delays expire. This thread sleeps during the time period given by the propagation delay of the first-ranked update in *outgoing*. When the sleeping period elapses, it sends to the peers all the updates whose propagation delay is equal to that time period, together with their unpropagated predecessors.

The propagation protocol for an update a is called after the successful execution of a at its initiator and it works in two steps. Firstly, the Propagator obtains the propagation delay associated to a by querying the StateMonitor. Secondly, if the delay is zero, then the update a is sent immediately to the peers, together with the local unpropagated updates that precede it. Otherwise, the Propagator inserts a on the right position in *outgoing*, respecting the ranking of the unpropagated updates in ascending order, by their propagation delays.

The Propagator provides the handler for executing on the local replica, a group of updates received from the peers (primitive onReceivedUpdates in Figure 4.5). The Propagator treats individually each received update a, as follows: if a already belongs to LocalHistory, it is ignored. If all the predecessors of a have already been applied locally, then a is executed and its identifier is inserted in LocalHistory. Otherwise the execution of a is deferred, by inserting it in *incoming*. Finally, the Propagator executes the updates in *incoming*, for which all the predecessors have been executed on the local replica.

The update propagation protocol can be easily extended so as to consider asynchronous conditions, that state if, at a given time instant, an update should be propagated to peers or not, according to the state of the current replica. Such conditions should be satisfied each time when the replica state changes (i.e. after the execution of a newly issued update), for each unpropagated update.

4.5 The scheduling protocol

Before being integrated into the Local History of each replica, an access passes through two main states: PENDING and ACCEPTED/REJECTED. An access is in the state PENDING while waiting for its scheduling decision. This decision is the result of a distributed scheduling protocol, to which cooperate a subset of replicas, called *schedulers*. An acceptance decision leads the access to the state ACCEPTED, and also contains its execution order with respect to the non-commutative accesses, issued concurrently. A rejection decision leads the access to the state REJECTED. We design a Scheduler object, whose interface (Figure 4.6) contains the handler for processing a scheduling request, the handler for processing a scheduling reply, and the primitives resolveBefore and resolveAfter, performing the scheduling before, respectively, after the access execution at its initiator. The primitive resolveBefore is used when the concurrency control mode is pessimistic. The primitive resolveAfter is used when the concurrency control mode is optimistic. The following two sections describe the protocols corresponding to the two concurrency control modes.

```
enum AccessState {PENDING, ACCEPTED, REJECTED}
class SchedulingRequest { };
class SchedulingReply { };
class Scheduler {
  ReplicaIds schedulers;
 public:
  virtual short onSchedulingRequest(SchedulingRequest* request)=0;
  virtual short onSchedulingReply(SchedulingReply* reply)=0;
  virtual short scheduleBefore(Access& a) { return OK; }
  virtual short scheduleAfter(Access& a) { return OK; }
};
```

Figure 4.6: The interface of the Scheduler

4.5.1 The pessimistic concurrency control protocol

In the case of the pessimistic concurrency control mode, an access remains in the state PENDING while waiting for the execution permissions from the *schedulers*. If all *schedulers* replied favorably, then the access reaches the state ACCEPTED. If at least one *scheduler* rejects the access execution, then the access reaches the state REJECTED.

The pessimistic scheduling protocol, performed by an object PessimisticScheduler, specialized from Scheduler (Figure 4.7), provides two main guarantees: if two updates are exclusive, then at most one of them is accepted and an access is accepted only if all its predecessors are already accepted. The protocol relies on a modified version of the Ricart-Agrawala mutual exclusion algorithm [57]. We enrich the original algorithm with the resolution policies between two conflicting or non-commutative accesses, as contained within the Contract.

The protocol uses the following data: the list *pending* of the locally initiated accesses, that are in the state PENDING, the list *accepted* of the locally initiated accesses, that are in the state ACCEPTED and a list of scheduling requests *deferredReplies*, to which the replies have been deferred. The PessimisticScheduler defers a reply, when it determines (at least) one predecessor in *pending*, for the access for which the execution permission has been required.

The PessimisticScheduler encapsulates the scheduling of an access a within the primitive resolveBefore, which works in two main steps. Firstly, the initiator of a sends to each *scheduler* a scheduling request for a (including to itself, if

```
class SchedulingRequest_p : public SchedulingRequest {ReplicaId initiator, Access a};
class SchedulingReply_p : public SchedulingReply {ReplicaId r, AccessId a, AccessState state, AccessIds preds};
class PessimisticScheduler : public Scheduler {
  GroupAccesses pending, accepted;
  vector<SchedulingRequest_p> deferredReplies;
};

short scheduleBefore(Access& a) {
  SchedulingRequest_p request = {a.initiator, a}
  a.state = PENDING; a.nb_replies=0
  pending = pending ∪ {a}
  send(schedulers, request)
  wait (a.state ≠ PENDING)
  pending = pending \ {a}
  for each e ∈ deferredReplies
    if (a ∈ e.reply.preds) {
      if (a.state == REJECTED) e.reply.preds = e.reply.preds \ {a.id}
      send e.reply for e.request if (∀ a' ∈ e.reply.preds, a'.state == ACCEPTED)
    }
  if ((a.state == REJECTED) or (a.state == TIMEOUT))
    return ERROR
  return OK
}

short onSchedulingReply(SchedulingReply_p* reply) {
  a = reply->a
  switch (reply->state) {
  case REJECTED: a.state = REJECTED; notify(a.state ≠ PENDING);
  case ACCEPTED: a.preds = a.preds ∪ {(reply->r, reply->preds)}
    a.nb_replies++
    if (a.nb_replies == |schedulers|) {
      a.state = ACCEPTED; accepted = accepted ∪ {a}; notify(a.state ≠ PENDING)
  }}
}

short onSchedulingRequest(SchedulingRequest_p* request) {
  r = request->r; a = request->a;
  reply REJECTED to request if:
   ∃ a' ∈ pending, so that (contract->detectAndResolve(a, a') == <conflicting, a'>) or
   ∃ a' ∈ accepted, so that (contract->detectAndResolve(a, a') ∈ {<conflicting, a or a'>, <non-commutativity, a>})

   preds = {a' ∈ pending ∪ accepted, where contract->detectAndResolve(a, a') == <non-commutativity, a'>}
  reply = {ACCEPTED, preds}

  defer reply, i.e. deferredReplies = deferredReplies ∪ {request, reply} if:
   ∃ a' ⊆ pending, so that (contract->detectAndResolve(a, a') == <non-commutativity, a'>)

  otherwise, send reply
}
```

Figure 4.7: The PessimisticScheduler

it is a *scheduler*). The primitive blocks until the PessimisticScheduler is able to take the scheduling decision for a. The decision combines the scheduling replies received from the *schedulers*. Each received reply is processed by the handler onSchedulingReply. If this is the first negative reply, then the state of a becomes REJECTED. Otherwise, if all replies are positive, then the state of a becomes ACCEPTED. Secondly, the PessimisticScheduler sends the reply for each request in *deferredReplies*, for which a has been the last predecessor in the PENDING state.

Any scheduling request is processed at a *scheduler*, by using the primitive onSchedulingRequest, which works as follows. The peer rejects the access a contained in the request, if one of the following three situations occur:

1. There are local pending updates with which a is exclusive, and the resolution policy requires to exclude a;
2. There are local accepted updates with which a is exclusive;
3. There are local accepted updates for which a is detected as predecessor;

If neither of the above three situations occur, then a is locally accepted. The scheduling reply for a also contains the predecessors of a. These are determined from the lists *pending* and *accepted*, according to the resolution policies for ordering two non-commutative accesses. If at least one predecessor of a is in the state PENDING, then the reply is deferred. Otherwise, the reply is returned immediately to the requestor.

The complexity of this algorithm, in number of messages exchanged among the participants, is $2{*}n$, where n is the number of schedulers.

4.5.2 The optimistic concurrency control protocol

In the case of the optimistic concurrency control mode, the reconciliation protocol, performed by the Reconciliator object, specialized from the Scheduler (Figure 4.8), decides the acceptance or the rejection of an update, a-posteriori with its tentative execution. An acceptance decision also includes the execution order of that update (i.e. the set of updates that must precede it).

The protocol uses the following data: the set *global_tentative* containing the tentative updates (issued locally or received from peers), the list *tentative_history* containing the identifiers of the tentative updates, executed on the current replica.

The reconciliation protocol is encapsulated within the primitive resolveAfter, that is called for an access a, after its tentative execution. It works as follows. Firstly, the Reconciliator inserts the identifier of a, in the LocalHistory object. Secondly, it queries the StateMonitor object, in order to determine if there are local tentative updates that need the scheduling decision immediately. The answer depends on the frequency of reconciliations (as given by the stabilization delay) or on the bound on the order error. If the reconciliation of tentative updates is needed, the Reconciliator asks its peers for the permission to compute the next schedule, by using the Ricart-Agrawala mutual exclusion algorithm. In

```
class SchedulingRequest_o : public SchedulingRequest { ReplicaId& r, LocalHistory& received};
class SchedulingReply_o : public SchedulingReply { ReplicaId& r, bool decision, GroupAccesses&
A};
class Reconciliator : public Scheduler {
  GroupAccesses global_tentative;
  LocalHistory tentative_history;
  bool coordinator;
 public:
      Reconciliator() {run reconciliate() with the frequency given by state_quality-
>getStabilizationDelay();}
};

short scheduleAfter(Access& a) {
  if a is executed locally, tentative_history=tentative_history ∪ {(a.initiator, a.id)}
  global_tentative = global_tentative ∪ {a}
  if (state_quality->isReconciliationNeeded())
    reconciliate();
}

void reconciliate() {
  nb_replies = 1
  send scheduling request, piggyback identifiers of updates in global_tentative
  wait (coordinator==true) or (nb_replies == 0)
  if (coordinator) {
    decided = getSchedule(); send(decided); coordinator = false
  }
}

short onSchedulingRequest(SchedulingRequest_o* request) {
  reply REJECTED, if (coordinator) or ((nb_replies > 0) and (crt_replica < request->r))
  else reply ACCEPTED with A={a ∈ global_tentative, a.initiator == crt_replica, a.id ∉ request-
>received}
}

short onSchedulingReply(SchedulingReply_o* reply) {
  if (reply->decision==ACCEPTED) {
    global_tentative = global_tentative ∪ reply->A
    nb_replies++
    if (all replies received) notify(coordinator==true)
  } else {
    coordinator = false; nb_replies = 0; notify(nb_replies == 0);
  }
}
```

Figure 4.8: The reconciliation protocol

```
void onReceiveDecided(GroupAccesses& A, GroupAccesses& R) {
  for each a ∈ R {
    if (a ∈ tentative_history) undo(a);
    global_tentative=global_tentative \ {a}
     if (a.initiator == crt_replica) { state_quality->notify(access_scheduled, a); send result of a to
a.caller
  }}
  for each a ∈ global_tentative, from the last to the first
    if (a ∈ tentative_history) undo(a);
  for each a ∈ A {
    if (a.preds) {
      if (a ∈ tentative_history) undo(a)
      if (a.preds ⊆ history) {
        wrapper->execute(a); history=history ∪ {(a.initiator, a.id)}
      } else incoming = incoming ∪ {a}
    } else {
      if (a ∉ tentative_history) wrapper->execute(a);
      history=history ∪ {(a.initiator, a.id)}
   }
   global_tentative=global_tentative \ {a}
   if (a.initiator == crt_replica) {
     contract->notify(access_scheduled, a); send result of a to a.caller
  }}
  for each a ∈ global_tentative, from the first to the last
    if (a_i ∈ tentative_history)
      if (wrapper->execute(a) != OK) tentative_history=tentative_history \ {(a.initiator, a.id)}
}

void onReceiveTentative(GroupAccesses& U) {
  for each a ∈ U {
    if (wrapper->execute(a)==OK) {tentative_history=tentative_history ∪ {(a.initiator, a.id)}};
    global_tentative=global_tentative ∪ {a}
  }
}
```

Figure 4.9: Processing updates

```
PairGroups* getSchedule() {
  refused = ∅
  for i = 2 to n
    for j = 1 to i - 1 {
      result = contract->detectAndResolve(a_i , a_j)
      if (!result) continue;
      switch (result) {
      case <non-commutativity, a_i>: a_j.preds=a_j.preds ∪ a_i
      case <non-commutativty, a_j>: a_i.preds=a_i.preds ∪ a_j
      case <conflicting>: a_i.exclusive=a_i.exclusive ∪ a_j; a_j.exclusive=a_j.exclusive ∪ a_i;
      case <conflicting, a_i>: refused=refused ∪ a_j; global_tentative=global_tentative \ {a_j};
      case <conflicting, a_j>: refused=refused ∪ a_i; global_tentative=global_tentative \ {a_i}; j=i;
      case <conflicting, a'>: refused=refused ∪ {a_i, a_j}; a_{n++}=a';
        global_tentative=global_tentative \ {a_i, a_j}; j=i
      }
    }

  get a' ∈ global_tentative, s.t. |a'.exclusive|=max{|a_i.exclusive|, i=1, n
  while (|a'.exclusive|) {
    refused=refused ∪ {a'}; global_tentative=global_tentative \ {a'}
    ∀ a" ∈ global_tentative, a".exclusive=a".exclusive \ {a'}
    get a' ∈ global_tentative, s.t. |a'.exclusive|=max{|a_i.exclusive|, i=1,n
  }
  check and break cycles in the precedence graph
  return (global_tentative, refused)
}
```

Figure 4.10: Computing a schedule

this respect, it sends to its peers a request, wherein it includes the identifiers of the updates in *global_ tentative*. Then, it blocks until a negative reply is received or until all needed replies are received.

A reply is processed by the primitive onSchedulingReply, as follows. If it is a rejection reply, the Reconciliator cancels the scheduling initiative. If it is an acceptance reply, the Reconciliator inserts the newly received updates in *global_ tentative* and increments the number of replies. If all replies are positive, the Reconciliator computes the schedule, by reconciliating all the updates in *global_ tentative*.

The Reconciliator doesn't initiate a reconciliation, if it has replied favorably to the coordination request coming from a peer. Also, the Reconciliator postpones any update initiated while there is a reconciliation in progress.

A request, for coordinating the next schedule, is processed by the primitive onSchedulingRequest, as follows. The request is rejected if the current replica has already became coordinator or it wishes to become coordinator and its priority is larger than that of the requester replica. Otherwise, it sends an acceptance reply, wherein it includes the local updates not yet received by the requester replica.

The computation of the schedule, performed by the primitive getSchedule (Figure 4.10), has three main steps. Firstly, for each access a in *global_ tentative*, the Reconciliator applies the conflict resolution policy from the Contract. If these policies don't require explicitly to reject a, then the Reconciliator computes the list of the predecessors of a (noted $a.preds$) and the list of accesses with which it is exclusive and the choice of the victim is left to the system (noted $a.exclusive$). Secondly, it tries to minimize the number of rejected accesses, by excluding iteratively from *global_ tentative*, the access a with the biggest number of exclusive relations (similar to B-IceCube algorithm [62]). Finally, it breaks the cycles in the precedence graph containing all accepted updates, if they occurred. The complexity of the reconciliation algorithm, in number of messages exchanged, is $3 * n$, where n is the number of replicas.

The Reconciliator provides the handler for processing a schedule, received from the coordinator (primitive onReceiveDecided in Figure 4.9). The schedule contains two groups, containing the updates accepted, respectively the updates rejected. Each rejected update, that belongs to *tentative_ history*, is undone. As in Bayou, each accepted update, that has at least one predecessor, is redone before all the tentative updates. Its execution occurs immediately, if all its predecessors belong to the LocalHistory. Otherwise, its execution is deferred. If the initiator of the scheduled update is the current replica, then the Reconciliator sends the invocation result to the caller.

The Reconciliator also provides the handler for processing the reception of a group of updates in the state PENDING. The Reconciliator tries to execute each update in the group, and includes its identifier in *tentative_ history*, if the tentative execution has been successful.

4.6 Resolving the dependencies

Obtaining the dependency set The dependency relationships between accesses newly issued and updates already accepted, include existing guarantees (e.g. Bayou's session guarantees) or they are expressed by means of predicates on the access attributes. We define the DependencyMonitor object, that manages the sets of accepted updates, called *dependency sets*, on which new accesses depend. It provides two main methods, that include a new access in the current dependency set and to obtain the identifiers of the updates contained in the dependency set.

We specialize the DependencyMonitor in order to represent the session guarantees of Bayou and Lamport happens-before. Precisely, in the case of the session guarantees of Bayou, the *dependency set* is common to all accesses with a given caller, and it is defined for each guarantee, as follows. If RYW is required, the *dependency set* contains the updates initiated by a given caller, at any replica. If MW is required, the *dependency set* contains the updates seen by the caller's last update. If *MR* or *WFR* is required, the *dependency set* contains the updates observed by the caller's last read. In each case, the dependency set is maintained client-side, replica-side or in a central repository. In the case of Lamport happens-before, the *dependency set* contains all the updates "known" by the current replica. The "known" updates include the updates issued locally and the updates received from the peers, and the dependency sets of those updates [48].

Obtaining the predecessors We define the resolver DependencyResolver that guarantees that the predecessors of each access are executed before that access (within the primitive resolve in Figure 4.11). The predecessors include the set of predecessors updates required by the caller and the *dependency set* obtained from the DependencyMonitor object. In this respect, the DependencyResolver pulls the predecessors that don't belong to the LocalHistory, from their initiators.

4.7 The overall consistency protocol

4.7.1 The replicated access execution

The overall consistency protocol for an access a is encapsulated within the method replicatedAccess of the ConsistencyManager (Figure 4.12). The method works similarly, both in the cases when the concurrency control mode is pessimistic and when it is optimistic. The method has seven main steps. In the first step, the ConsistencyManager creates an Access object a from the object Invocation, passed as argument. The identifier of a contains the initiator identifier and the current number of invocations to the called operation. The second step depends on the concurrency control mode. If this is optimistic, then scheduleBefore returns OK. If the concurrency control mode is pessimistic, then the Scheduler determines if a is accepted and what is its execution order. In the third step, the object DependencyResolver is invoked in order to guarantee that

```
class DependencyMonitor {
  virtual Predecessors getDependencySet(Access& a)=0;
  virtual void updateDependencySet(Access& a)=0;
};

class DependencyResolver {
  DependencyMonitor* resolver;
 public:
  short resolve(Access& a);
  short waitPreds(Access& a);
};

short resolve(Access& a) {
  dep_set = resolver->getDependencySet(a)
  if (dep_set) a.preds = a.preds ∪ dep_set
  for each peer r with an entry ∈ a.preds {
    absent_ids = ∅
    for each id ∈ a.preds[r]
      if (id ∉ history[r]) absent_ids = absent_ids ∪ {id}
    if (absent_accs ≠ ∅)
     if ((code = getUpdates(r, absent_ids)) == ERROR)
       return ERROR
  }
  return OK
}
```

Figure 4.11: The DependencyResolver

the predecessors of a have already been integrated into the current replica state. In the fourth step, the Propagator is invoked in order to guarantee that the local state that will be observed by a satisfies the predicates maintained by the StateMonitor. In the fifth step, the access a is executed locally, by calling the Server-Side Replica Wrapper. In the sixth step, if the access a is an update, then the Propagator is invoked, so as to enforce the visibility of a at all peers. The seventh step also depends on the concurrency control mode. In the pessimistic case, it returns OK. In the optimistic case, the Scheduler is invoked in order to resolve the potential conflicts or wrong ordering induced by a.

4.7.2 Proving the correctness of the consistency protocol

In this section, we prove that the consistency management protocol provides the constraints required for each access.

4.7.2.1 Notations and assumptions

We note a the access for which the protocol is performed and $a.tmst$ the timestamp of the issuance of a. For simplicity reasons, we neglect the scheduling delays and the invocation delays. We consider the communication delays bounded, but the underlying bound, noted *comm_ delay*, is unknown.

We specify the scheduling relations between accesses, using the formalism ACF (Actions-Constraints Framework) [62]. In this formalism, the relation a' precedes a is represented by $a' \rightarrow a$. The mutual exclusiveness between two actions a and a' is represented by a cycle: $a \rightarrow a' \rightarrow a$. If the victim is a, this

```
class ConsistencyManager {
  DependencyResolver* acceptor;
  Scheduler* scheduler;
  Propagator* propagator;
  ServerSideReplicaWrapper* wrapper;
};

short replicatedAccess(Invocation& call) {
  Access* a = createAccess(call, history->getCallCount(call.id.op), contract->getWeight(call))
  short code = OK;
  if (contract->needsScheduling(*a)) {
    code = scheduler->scheduleBefore(*a);
    if (code == OK) scheduling_needed = true;
  } else a.state = ACCEPTED;

  if (code == OK) {
    code = acceptor->resolve(*a);
    if (code == OK) {
      propagator->makeObservable(*a);
      code = execute(*a);
      if ((code == OK) and (contract->isUpdate(*a))) propagator->propagate(*a);
      if (scheduling_needed) code = scheduler->scheduleAfter(*a);
    }
  }
  return code;
}

short execute(Access& a) {
  short code = wrapper->execute(a);
  if (code == OK) {
    history = history ∪ {(crt_replica, a.id)}
    acceptor->updateDependencySet(a);
  }
  return code
}
```

Figure 4.12: The replicated access execution

is represented by $a \rightarrow a$. The execution of a' before a on the current replica is represented by a' $<_s$ a.

4.7.2.2 The protocol's properties

The correctness of the protocol contains a *liveness* and a *safety* property, that have to be proved for each access. The *liveness* property is defined as follows: each access is applied all over (i.e. at all replicas, if it is an update, and only at its initiator, if it is a read) or rejected in a finite amount of time.
The *safety* property is defined as follows: for each access, the protocol satisfies at all the replicas, the consistency constraints associated to that access.

We note the consistency constraints attached to the access a by $C(a)$. This is a structure formalized as follows:
{*numerical_error*, *order_error*, *propagation_delay*, *stabilization_delay*, $C(a, a')$, *precedence_condition*}. All elements in this structure are optional.

$C(a, a')$ defines the scheduling relation between a and the concurrent access a', among the following possible options:

- a and a' are commutative and non-conflicting: $a \| a'$;
- a and a' are conflicting, where the access to exclude is a or a': $a' \rightarrow a \rightarrow a' \wedge (a \rightarrow a \vee a' \rightarrow a')$;
- a and a' are not commutative, where the predecessor is a or a': $a \rightarrow a' \vee a' \rightarrow a$;

In summary:

$$C(a, a') = \begin{cases} a \| a' \\ a' \rightarrow a \rightarrow a' \wedge (a \rightarrow a) \\ a' \rightarrow a \rightarrow a' \wedge (a' \rightarrow a') \\ a \rightarrow a' \\ a' \rightarrow a \end{cases}$$

The *precedence_condition* requires that the predecessors of a, noted $a.preds$, are executed before a at all replicas. Some of these predecessors are known at the issuance time of a, the others are determined during the scheduling of a. More formally, the *precedence_condition* guarantees that:
$\forall a' \in a.preds : a' <_s a$.

The proof is built in three steps. The first step identifies the states through which an access passes during its lifetime, together with the transitions. The second step decomposes $C(a)$ into local conditions, that trigger the transitions (at the initiator or at a peer) or that are used by the transitions. The third step proves the *liveness* and *safety* properties for each intermediate state. The *liveness* property guarantees that the access reaches the intermediate state in a finite amount of time. The *safety* property guarantees that the transition towards that state satisfies the local conditions from $C(a)$, that it is concerned with.

4.7.2.3 The consistency state machine

Access states We call the states through which an access passes during its lifetime: *issued*, *pending locally*, *accepted*, *rejected*, *deferred locally*, *ready*, *applied locally*, *outgoing*, *sent*, *received*, *deferred remotely*, *applied remotely*, *waiting acks*, *terminated*.

An access is in the state *issued*, after being submitted by its caller at its initiator. The states *pending*, *accepted* and *rejected* have been defined within the concurrency control, described in section 4.5. "Locally" means that the state is met at the initiator of the access, and "remotely" means that the state is met at a peer. An access is *deferred* for execution, when at least one of its predecessors hasn't been applied on the current replica. After all its predecessors have been applied on the current replica, the access reaches the state *ready*. An access reaches the state *applied*, after being executed definitively on the current replica (i.e. its execution will never be undone). An access reaches the state *outgoing*, while it is waiting for its *propagation delay* to elapse. An access reaches the state *sent* at its initiator, after being propagated to the peers. When a remotely issued access arrives locally, it is in the state *received*. An access is in the state *waiting acks* at its initiator, while it is waiting for acknowledgements that the access has been successfully applied at each peer. If the acknowledgements are not needed, then the access reaches directly the final state *terminated*.

The local history of the replica r_i is formalized, as follows:
$LH^i = \{a/(a.state = appliedlocally) or (a.state = appliedremotely)\}$.

For each access, the consistency protocol is represented by a distributed state machine, representing the access execution at its initiator and, in the case of updates, at each peer.

State transitions The transitions between successive states invoke one of the three resolvers or read the Contract. The transition from the state *deferred* (*locally* or *remotely*) is provided by the DependencyResolver. The transition from the state *pending locally* is provided by the PessimisticScheduler or by the Reconciliator (according to the concurrency control mode *pessimistic* or *optimistic*), the transition from the state *outgoing* is provided by the Propagator. The transitions from the states *issued* and *outgoing* consult the consistency constraints attached to the access (i.e. C(a)), the transitions from *accepted*, *applied locally* and *received* consult the attributes of the access. The transition from *waiting acks* is immediate, if all the acknowledgements were received. The transition from the state *ready* to the state *applied* (*locally* or *remotely*) executes the access on the current replica.

Figure 4.13 shows the sequence of states within the state machine representing the consistency protocol for an access. In order to shorten the transition names, we replaced ServerSideReplicaWrapper::execute by execute, DependencyResolver::resolve by getExecPreds, DependencyResolver::waitPreds by waitExecPreds, Propagator::send by send, PessimisticScheduler::resolveBefore and Reconciliator::resolveAfter by schedule.

The *liveness property* is refined as follows: each access reaches its final state:

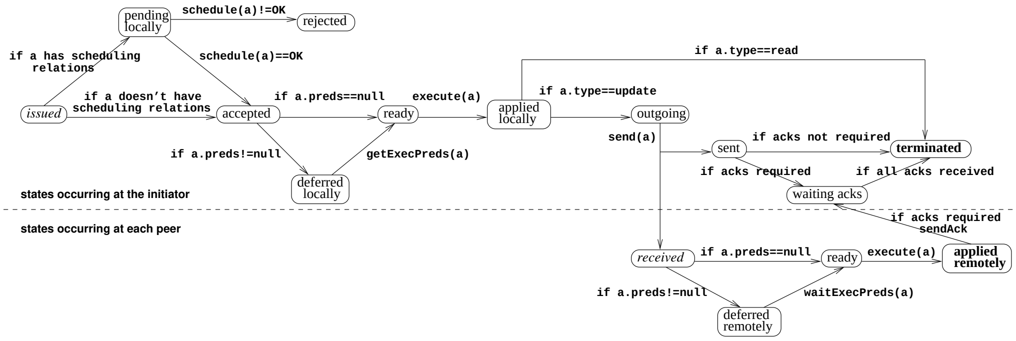

Figure 4.13: The consistency state machine

terminated or *rejected*, at the initiator, and *applied remotely*, at the peers, in a finite amount of time. This is equivalent to:
$\forall a$, $\exists$ *period*, s.t. at the time instant $t, t = a.tmst + period$, $a \in LH^i$, $i = 1, n$ or $a.state$=rejected.

The *safety property* is formalized as follows: $\forall$ a, $C(a)$ is satisfied at all the replicas.

Local conditions inferred from the Contract The consistency constraints $C(a)$, associated to the access a, are decomposed into two categories of local conditions. The former category includes the local events, that trigger the transitions between successive states, met at the initiator or at a peer. The latter category includes the scheduling relationships and the precedence to be satisfied for each access. The two categories are detailed in the next two sections.

Transition events A transition event prevents an access from being blocked forever in a particular state. For the access a, the transition events include: the propagation delay of a has elapsed, the local bound on numerical error has been exceeded, the bound on the number of local pending accesses has been exceeded, the stabilization delay of a has elapsed, all the predecessors of a have been executed. We associate to these events five conditions, formalized as follows:

- $a.state == outgoing$ during $period \ \leq \ C(a).propagation\_delay$ (C1);
- $\Sigma_{\ a'/a'.state==outgoing}\, a'.weight \ \leq \ C(a).numerical\_error/(n-1)$, where n is the number of replicas, $n > 1$ (C2);
- $|a'/a'.state ==$ *pending locally* $\ \leq \ C(a).order\_error$ (C3);
- $a.state ==$ *pending locally* during $period \ \leq \ C(a).stabilization\_delay$ (C4);
- $\exists\ a' \ \in a.preds$, $a'.state \ \neq$ *applied remotely* (C5);

The condition (C1) states that the access a remains in the state *outgoing* at most during the period given by $C(a).propagation\_delay$. The condition (C2) states that, at each replica, the total weight of the updates in the state *outgoing* is bounded by $C(a).numerical\_error/(n-1)$. The condition (C3) states that, at each replica, the number of the updates in the state *pending* is bounded by $C(a).order\_error$. The condition (C4) states that the access a remains in the state *pending locally* at most during the period $C(a).stabilization\_delay$. The condition (C5) states that at least one predecessor of the access a is missing from the current history. A transition event occurs when the corresponding condition isn't respected.

We note by $max\_propagation\_delay$ the maximum propagation delay computed over all updates. More formally:
$max\_propagation\_delay = max\{C(a).propagation\_delay / \ \forall \ a\}$.

Scheduling relationships The scheduling concerns only the concurrent accesses. We consider two accesses a and a' as concurrent, if $\exists$ the time instant t, s.t. at t, both a and a' have already been issued, but $a.state! = appliedremotely$ at $a'.initiator$ and $a'.state! = appliedremotely$ at $a.initiator$.

The scheduling mechanism should respect the following condition, noted (C6): $\forall a$ and a' two concurrent accesses, a is compared with a' at the initiator of a and at the initiator of a' or at the *coordinator* replica. The comparison respects $C(a, a')$, if a and a' are both in the state *pending* at the comparison time. Otherwise, if a is in the state pending, while a' is in the state accepted, then a will be rejected, in the case of conflicts or in the case of non-commutativity, where it is the predecessor. The case where a' is in the state pending and a is in the state accepted is similar.

Access classes According to the possible consistency constraints within $C(a)$, we identified five main access classes. These include:
1) reads that don't need concurrency control
2) reads that need pessimistic concurrency control
3) updates that don't need concurrency control (as they don't have any scheduling relationship)
4) updates that need pessimistic concurrency control
5) updates that need optimistic concurrency control

Each class contains a second classification level, representing: i) the accesses without any predecessors and ii) the accesses that have predecessors. Each class has attached its particular chaining of pertinent states.

For the access a, we prove that the *liveness* and the *safety* properties are satisfied, when reaching each intermediate state. We prove the *liveness* property by showing the transition event(s) and by bounding the transition delay. The *safety* property is defined only if the underlying transition is triggered by any of the local conditions, identified above. If this is the case, we show that these conditions are satisfied.

4.7.2.4 Reaching *pending locally*

The *pending locally* state is met for any access that belongs to the classes 2), 4) or 5). The transition to *pending locally* is triggered immediately, its delay is zero, and it doesn't have to fulfill any consistency constraints. If the concurrency control mode is optimistic, the *pending locally* state is preceded by the tentative execution of a.

4.7.2.5 Reaching *deferred locally*

The *deferred locally* state is met for any access with predecessors. The transition to this state is triggered immediately, its delay is zero, and it doesn't have to fulfill any consistency constraints.

4.7.2.6 Reaching *accepted* or *rejected*

The *accepted* state is provided by the transition schedule for any access in the classes 2), 4) or 5) and is implicit in the other cases. We have to prove that the transition schedule satisfy the *liveness* and the *safety* properties. The *liveness* property requires that any access is accepted or rejected in a finite amount of time. The *safety* property contains the scheduling condition (C6).

The case of the pessimistic concurrency control If the access a belongs to the classes 2) or 4), then the transition schedule is called immediately after the issuance of a, by its initiator, noted r. It relies on the processing of scheduling requests and the associated replies at the *schedulers*, as shown in section 4.5.1.

We have to prove that a becomes *accepted* or *rejected* at r, in a finite amount of time and that it is compared with all its concurrent accesses, so as to fulfill $C(a, a')$. We construct the proof in three incremental steps. In the first step, we consider only one *scheduler* r' and at most one access a' concurrent with a, where a' has been issued at the replica r'. In the second step, we consider several accesses concurrent with a at the only *scheduler* r'. In the third step, we consider several *schedulers*, each one containing several accesses concurrent with a. In each step, we study the processing of the scheduling request (noted Q_a) by r', and the processing of the corresponding reply (noted A_a), as shown in the following.

I. One scheduler, one concurrent access
In the first step, we distinguish between the following cases, that could occur when Q_a arrives at r':
1) a' is in the state pending
1.1) $Q_{a'}$ has been sent to r
1.1.1) $A_{a'}$ has been received from r
1.1.2) $A_{a'}$ hasn't been received yet
1.1.2.1) $A_{a'}$ has been sent by r before Q_a
1.1.2.2) $A_{a'}$ has been sent by r after Q_a
1.1.2.3) $A_{a'}$ hasn't been sent yet

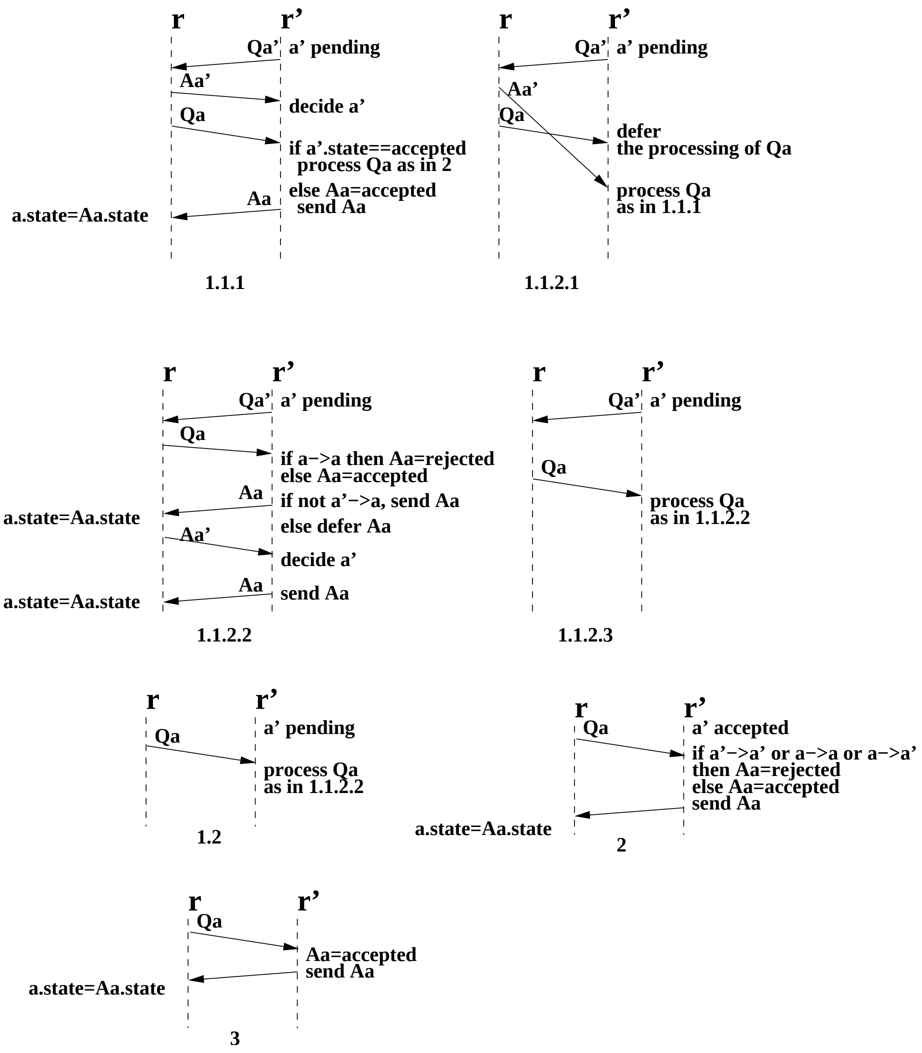

Figure 4.14: Scheduling with 1 scheduler and 1 concurrent access

1.2) $Q_{a'}$ hasn't been sent to r
2) a' is accepted (but r hasn't received it yet)
3) $\nexists a'$ at r', a' concurrent with a

Figure 4.14 shows how the acceptance decision for a is taken in each case. In the cases 1.1.1 and 2. r' handles Q_a after deciding the acceptance of a'. if a' is accepted, and a and a' are not conflicting and a isn't the predecessor of a', then r' accepts a'. Otherwise, r' rejects a. In both cases, the corresponding reply A_a created by r' is sent to r.

In the case 1.1.2.1, r' defers the processing of Q_a until the reception of $A_{a'}$. At this point, r' decides a' and then it processes Q_a as above.

In the cases 1.1.2.2, 1.1.2.3 and 1.2, a' is still pending, when r' processes Q_a. If a' is not the predecessor of a, then r' sends immediately the acceptance/rejection reply A_a to r. Otherwise, it defers the dissemination of the acceptance reply to r, until a' becomes decided.

In the case 2, r' accepts a and sends immediately A_a to r.

r takes the scheduling decision for a immediately after receiving A_a. If A_a rejects a, then a reaches the final state *rejected* and it isn't propagated to the peers. Otherwise, a reaches the state *accepted* and *a.preds* are extracted from A_a. As A_a could be deferred at r', its scheduling decision is bounded by the parameter *timeout*, which is system-defined or equal to $C(a).stabilizationdelay$. If A_a isn't received before *timeout*, then a is rejected.

A timeout could occur when the scheduling produces a deadlock. This deadlock is due to a cycle in the graph with the precedence relations between the pending accesses. Without the timeout parameter, none of these accesses could be decided (as accepted or rejected). In fact, any of these accesses is waiting for at least a scheduling reply that it is never delivered (as it has been deferred because of a pending access, that is never decided, as it is also waiting for a scheduling reply and so on). The timeout parameter could have different values for different replicas, so as to reflect their priorities. A bigger value for the timeout increases the chance for the accesses issued at that replica to be accepted, in the case of the cycle in the precedence graph.

In conclusion, a becomes accepted or rejected at its initiator in a finite amount of time, bounded by the *timeout* value associated to r.

II. Several concurrent accesses
In the second step, we consider at r' a set C, containing the accesses issued at r', and that are concurrent with a. Any access in C is either in the state pending or in the state accepted, but not yet received by r.

When Q_a arrives at r', it is processed according to the algorithm, depicted in Figure 4.15, so as to produce the output A_a. We define the union A_a of two replies A'_a and A''_a ($A_a = A'_a \cup A''_a$) as follows:
if A'_a==rejected or A''_a==rejected then A_a=rejected
else A_a=accepted and $A_a.preds = A'_a.preds \cup A''_a.preds$

```
A_a=∅
for each a' ∈ C{
  if a' is pending, then A'_a is computed by applying the case I.1)
  if a' is accepted, then A'_a is computed by applying the case I.2)
  A_a=A_a ∪ A'_a
  if A'_a.state == rejected, then send A_a to r
}
if a has pending predecessors, then defer A_a until those predecessors are accepted or rejected
else send A_a to r
```

Figure 4.15: Processing a scheduling request

After receiving A_a, r takes the scheduling decision for a, as shown in the previous step. The maximum delay before a reaches the state *accepted* or *rejected* has the same timeout value, as in the previous step.

III. Several schedulers
In the third step, we consider several schedulers r_i, $i = 1,\ m$. Each r_i determines the reply A_a^i by processing Q_a, as shown in the previous step. After receiving all the replies, r takes the scheduling decision A_a, by computing their union. More formally, $A_a = A_a^1 \cup ... \cup A_a^j$, where $j < m$, if A_a^j==rejected else j is m

r determines the state and the predecessors of a from A_a, as in the first step. The maximum delay for a to reach the state *accepted* or *rejected* has the same timeout value, as in the first step.

The case of the optimistic concurrency control In the case of an access that belongs to the class 5), the transition schedule is called, when one of the conditions (C3) or (C4) is violated. In both cases, the delay for triggering the transition schedule is given by the time period $C(a).stabilization\_delay$.

We show that the optimistic scheduling satisfies the *liveness* and the *safety* properties. The *liveness* property is satisfied, because a is accepted or rejected by the coordinator before the time period defined by:
$C(a).stabilization\_delay\ +\ timeout$ (where the value of *timeout* bounds the time period during which the replica waits for the permissions to become coordinator).
If a has been accepted, then it is applied immediately at r, as it is contained within the schedule which includes also all its predecessors, and the reconciliation mechanism makes sure that there are no cycles among the accesses accepted.

With respect to the *safety* property, we have to prove that a is compared with any concurrent access a' and the scheduling relation is applied at all replicas. We consider that a has been issued between two successive coordination requests, issued at the time instants t_sched_1 and t_sched_2, conducting respectively to the schedules S_1 and S_2. We consider that a' has been issued between two successive coordination requests $t\_sched'_1$ and $t\_sched'_2$, conducting to the schedules S'_1 and S'_2.
More formally: $t\_sched_1 < a.tmst \leq t\_sched_2$ and $a \in S_2$

$t\_sched'_1 < a'.tmst \leq t\_sched'_2$ and $a' \in S'_2$

We distinguish between three cases:
1) if $t\_sched_2 == t\_sched'_2 \Rightarrow a,\ a' \in S_2 == \mathrm{S}'_2$ and a and a' are compared by the coordinator respecting $C(a, a')$
2) if $t\_sched_2 < t\_sched'_2 \Rightarrow S_2$ is applied before S'_2 at all replicas $\Rightarrow a <_s a'$
3) if $t\_sched_2 > t\_sched'_2 \Rightarrow S'_2$ is applied before S_2 at all replicas $\Rightarrow a' <_s a$

4.7.2.7 Reaching *ready*

The transition to the state *ready* should also satisfy the *liveness* and the *safety* properties. The *liveness* property requires that the predecessors of a are applied at any replica in a finite amount of time. This property is decomposed into two sub-properties. The former sub-property requires that all the predecessors of a should arrive sooner or later at that replica. The latter sub-property requires that all the predecessors should be applied in a finite amount of time (i.e. there is no cycle in the precedence graph containing the accepted accesses). The *safety* property contains the *precedence_condition* from $C(a)$. The proof of these properties relies on the guarantee of our protocol, stating that when a is accepted, all its predecessors have already been accepted.

Predecessors reception The access a has a limited number of predecessors, because at any time there is a limited number of accepted accesses, to be propagated at all replicas. We distinguish between the direct and indirect predecessors of a, where the direct predecessors are determined by the scheduling of a, and the indirect predecessors are the predecessors of the direct or indirect predecessors of a.

The predecessors of a are obtained immediately, when they are requested by the initiator of a, by calling the transition getExecPreds. If the transition waitExecPreds is used, the delay to obtain the predecessors is bounded by:
$|a.preds| * (max\_propagation\_delay\ +\ comm\_delay)$

Predecessors execution Once all the predecessors are received, they can be executed if there is no cycle in the graph containing the precedence relationships among the accepted accesses. We have to prove that this graph doesn't contain any cycle. In order to prove this property, we suppose that there are n accepted accesses, so that their precedences produce a cycle. We note the accesses by a_1, a_2, ..., a_n and their acceptance times, respectively, by t_1, t_2, ..., t_n. The cycle as expressed as follows: $a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_n \rightarrow a_1$.
From $a_1 \rightarrow a_2 \Rightarrow t_1 < t_2$ (1)
From $a_2 \rightarrow a_3 \Rightarrow t_2 < t_3$ (2)
From (1) and (2) $\Rightarrow t_1 < t_3$
... We assume $t_1 < t_i$
From $a_i \rightarrow a_{i+1} \Rightarrow t_i < t_{i+1}$ (i), $i = 2,\ n-1$
From $t_1 < t_i$ and (i) $\Rightarrow t_1 < t_{i+1}$, $i = 2, n-1$
let $i = n \Rightarrow t_1 < t_n$

From $a_n < a_1 \Rightarrow t_n < t_1$ that contradicts $t_1 < t_n$

Another way to prove that a has a limited number of predecessors relies on the property that there are indirect predecessors of a that have no direct predecessors. In order to prove this property, we assume that any accepted access has at least one direct or indirect predecessor. We consider the set of all accepted accesses, noted a, and the set containing the successors of a, noted a', i.e. $\forall a_1 \in A' \subseteq A$, $a \rightarrow a_1$

Let a' be one predecessor of a ($a' \rightarrow a$). We distinguish between the following three cases: 1) $\{a'\} \cup a' == a$, 2) $a' \in a'$ and 3) $a' \in A \setminus A'$ and $|A \setminus A'| > 1$

In the case 1), $\exists a'' \in a'$ s.t $a'' \rightarrow a' \rightarrow a \rightarrow a''$. In other words, the precedence graph contains a cycle, that is impossible according to the previous result. In the case 2), the precedence graph also contains a cycle, because $a' \rightarrow a \rightarrow a'$. In the case 3), we include a'' in a' (i.e. $a' = a' \cup \{a''\}$), we set $a = a'$', and check again if the case 1), 2) or 3) occurred. This algorithm always stops by detecting the case 1) or 2), i.e. it detects a cycle in the precedence graph, that is impossible.

In conclusion, all the predecessors of a are applied within the time period given by:
$|a.preds| * (max\_propagation\_delay + comm\_delay)$
Providing $precedence\_condition$ from $C(a)$ is an intrinsic feature of the method resolve of the DependencyResolver.

4.7.2.8 Reaching *applied locally*

The state *applied locally* is met for any access. The transition to this state is triggered immediately, its delay is zero, and it doesn't have to fulfill any consistency constraints.

If the concurrency control is optimistic, the transition to *applied locally* may be absent. Precisely, if a has already been applied tentatively and it doesn't have new predecessors, it reaches the state *applied locally* directly from the state *ready*. If a has new predecessors, the action execute is preceded by undo.

4.7.2.9 Reaching *outgoing*

The *outgoing* state is met for any access in the classes 3), 4) or 5). The transition to this state is triggered immediately, its delay is zero, and it doesn't have to fulfill any consistency constraints.

4.7.2.10 Reaching *sent*

The *sent* state is met for any access in the classes 3), 4) or 5). The transition to this state is triggered when one of the two conditions (C1) or (C2) is violated. Its delay is bounded by the time period given by $C(a).propagation\_delay$.

4.7.2.11 Reaching *waiting acks*

The *waiting acks* state is met for any access, when the needed acknowledgements have been correctly received from the peers. The transition to this state is triggered immediately and its delay is zero.

4.7.2.12 Reaching *received*

The initial state *received* is met at a peer (concurrently with the state *sent*) within the period defined by:
$C(a).propagation\_delay\ +\ comm\_delay$.

4.7.2.13 Reaching *deferred remotely*

The properties of the transition to the *deferred remotely* state are treated similarly to the case of the *deferred locally* state.

4.7.2.14 Reaching *ready*

The properties of the transition to this state are treated similarly to the case of the *ready* state, met at the initiator.

4.7.2.15 Reaching *terminated*

The *terminated* state is met immediately from the *sent* state, or after receiving the acknowledgements from all the peers. The delay of reaching this state from the *sent* state is bounded by $(n\ -\ 1)\ *\ comm\_delay$.

In conclusion, the maximum delay for a to reach the final state *terminated*, from the moment when it has been issued, is:
$C(a).stabilization\_delay + timeout + |a.preds| * (max\_propagation\_delay + comm\_delay)\ +\ (n\ -\ 1)\ *\ comm\_delay$,
where *timeout* is the timeout value associated to r.

4.7.2.16 Reaching *applied remotely*

The properties of the transition to the *applied remotely* state are treated similarly to the case of the *applied locally* state.

The access a reaches the final state *applied remotely* within the time period given by:
$C(a).stabilization\_delay + timeout + 2 * |a.preds| * (max\_propagation\_delay + comm\_delay) +\ C(a).propagation\_delay\ +\ comm\_delay$,
where *timeout* is the timeout value associated to r.

4.7.3 Experimental evaluation of the consistency protocol overhead

We evaluated experimentally the overhead of the generic replica consistency protocol, integrated within the Replica Wrapper associated to the e-learning service. We compared the response time obtained for the invocations to the operation addCourse in the centralized case, with the response time obtained when the e-learning service is replicated. This comparison gives the cost of the calls to the method replicatedAccess of the Consistency Manager. This cost includes the time needed in order to reify the access and the time needed in order to invoke successively the three resolvers, composing the consistency protocol.

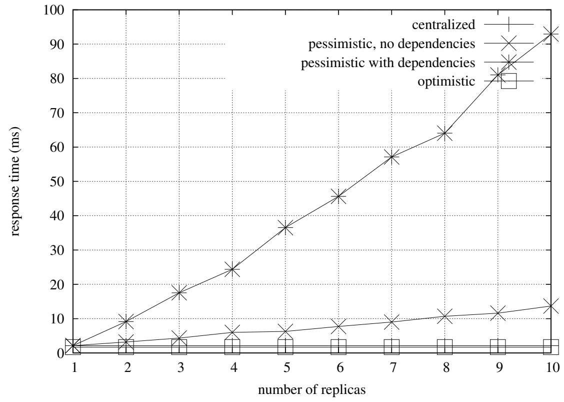

Figure 4.16: The replica consistency protocol overhead

We considered three consistency contracts. In the former contract, the concurrency control is pessimistic and updates have no dependencies. In the second contract, the concurrency control is also pessimistic, but updates have dependencies, among the updates of each peer. In the latter contract, the concurrency control is optimistic. We varied the number of replicas between 1 and 10, and we measured in each case the response time obtained by 22 invocations to addCourse, and we computed the average response time. We plot on the graph in Figure 4.16 the curves representing the average response time obtained by the invocations to addCourse, in the case of the three consistency contracts and in the centralized case. The curve corresponding to the optimistic contract is very close to the line (parallel with x-axis) corresponding to the centralized case. The curves corresponding to the pessimistic contracts show that the response time of the addCourse invocations increases linearly with the number of replicas. These experiments reveal two positive results. The former result shows that the overhead of the generic replica consistency management is insignificant when the concurrency control is optimistic. The latter result shows that the protocol's overhead increases linearly with the number of replicas when the concurrency control is pessimistic. This result leads us to the conclusion that the pessimistic concurrency control is pertinent when the number of replicas is small, and the replicas are fully-connected.

4.8 Conclusion

We developed a consistency protocol, capable to realize any consistency model, which could be instantiated from the meta-model. We prove formally that the protocol works correctly for each access, by means of the two properties liveness and safety. With respect to liveness, we show that each access is either applied at all replicas or rejected in a finite amount of time. With respect to safety, we show that the consistency constraints, associated to that access, are respected at all replicas. We also showed experimentally that the consistency protocol has a reasonable overhead, which includes the delay for creating the access, the delay of calling the client-side and server-side replica wrappers and the delay of executing the access at the replica that has been assigned to the client.

Part III

Response-Driven Replica Selection

Chapter 5

Survey of Response-Driven Replica Selection Systems

5.1 Introduction

We surveyed existing replication systems that select the suitable replica for each client, so as to improve the response time that will be experienced by the client. We classify the replica selection systems, according to the metric(s) that they use, in order to ameliorate the response time. We identified four classes: server load-based selection systems, network proximity-based selection systems, response time estimation-based selection systems and systems providing flexible selection criteria. The former class include systems like Akamai [15], Radar [53], [42] and Refresco [45], which find the suitable replica, at the beginning of the session or for each transaction, by optimizing the server load. The second class include systems like Radar [53], aCDN [54], Globule [49] which optimize the network proximity, quantified by means of network router hops or round-trip time. The third class include systems like Web++ [74], Leganet [22], AQuA [33], which optimize the estimated response time or impose a threshold on the estimated response time. The latter class include systems like anycast service [77], which use flexible replica selection policies combining several performance metrics.

The rest of this chapter is structured in five sections. Sections 5.2, 5.3, 5.4 present the systems selecting the suitable replica using, respectively, the replicas load, the network proximity and a response time estimator. Section 5.5 presents the systems providing flexible criteria for replica selection. Section 5.6 concludes the chapter.

5.2 Server load-based selection systems

RaDar (Replicator and Distributor and Redirector) [53] is a decentralized replication system that provides scalable access to replicated Internet services, while adding and deleting replicas dynamically. The replication decisions, such as request redirection and replica placement are resolved locally by each replica without the need of global knowledge about the system. It selects the suitable replica for each request, by optimizing the network proximity, while avoiding an overloaded replica. The network proximity is quantified in number of router hops. The replica load is quantified by the number of connections performed during a given time period. Their request distribution approach provides load balancing, by maintaining the ratio between the load of the most loaded replica and the load of the less loaded replica below a given threshold. Their approach improves response time by 13% compared to the centralized case. Their replica placement policy favor the hosts belonging to the network paths, by where most clients requests passed, when they were submitted to the client-assigned replica. Their replication approach provided 52% savings in bandwidth, compared to the centralized case.

aCDN (Application Content Delivery Network) [54] continues the authors' work performed within RaDar. One improvement, that they propose, concerns the request distribution strategy, wherein they avoid selecting a distant replica, when there are nearby overloaded replicas. They consider that the client hosts belong to (geographical or network) regions. They compute for each replica the probability of being selected for the clients which belong to a given region. The probability, assigned to a given replica, with respect to a given region, takes into account the current replica load, compared to an inferior and a superior threshold, and the distance between the replica and that region.

In Refresco [45], the secondary replica, chosen so as to execute a given query, is the one with the minimal cost. The system contains a query routing component, which evaluates the cost of executing the query, at each secondary replica, by taking into account the current load of the node and the execution times of the refresh transactions which must be applied before the query (in order to reach to freshness level required by the query). The load of a node is estimated by summing the remaining execution times of the currently running transactions (including the refresh transactions). The execution time of a transaction is estimated by using the response times previously obtained by executing the same transaction. The query execution is delayed so as to apply the sequence of refresh transactions, needed so as to satisfy the freshness requirements. They showed experimentally that relaxing the data freshness and the requirements granularity improve both query response times and throughput.

5.3 Network proximity-based selection systems

Globule [49, 50] is a user-centered CDN, which replicate transparently Web sites on world-wide spreaded hosts, in order to improve the sites' responsiveness and availability, as perceived by the clients. Globule exploits the distribution of client requests recorded in a given time period, in order to place new replicas dynamically on hosts located in regions from where most requests came from. In this respect, they place the clients in a M-dimensional geometric space, that they divide in regions of identical sizes. They rank the regions according to the number of clients that they contain, and place a new replica in the region with the position k in the rank [50].

The system contains a redirector component, which chooses automatically the suitable replica to which the clients requests are redirected, using a HTTP-based mechanism or a DNS-based mechanism. The former is applied for each requested page, while the latter is applied at the site-level. Both mechanisms are customized by a redirection policy, which by default optimizes the proximity between the client and the available replicas, where the proximity is defined in terms of network latency. The latency between the client and the replica is estimated as the Eucledian distance, computed using the positions of the corresponding nodes in a M-dimensional geometric space [50, 64]. Each binding of the client to a replica is associated a TTL (usually 10 minutes). This allows choosing the suitable replica dynamically. When using "Versatile anycast", it is possible to choose the suitable replica for each request.

5.4 Response time estimation-based selection systems

Carter et al. [7] were among the first to develop a response time-driven replica selection approach by optimizing dynamic metrics, like: round-trip time, available bandwidth, or the metric, called *PredictedTT*, which is derived from the previous two. They showed experimentally the benefits of dynamic metrics over the random policy or the static metrics, like the number of hops, in order to improve the client-perceived response time. Precisely, they found that the correlation between the round-trip time and the response time is 0.51. The *PredictedTT* metric estimates the transfer time of Web documents. It has a formula that sums the round-trip time and the transmission time, weighted, respectively, by two factors. The transmission time is the document size divided by the available bandwidth.

Chen et al. [9] also perform response time-driven replica selection approach, for Internet services (e.g. WWW, FTP). Their approach is integrated transparently within DNS and relies on estimating the response time of requests, by the metric called *Weighted Total Response Time*. This metric has a formula equal to *Predicted Transfer Time* multiplied by a weighting factor aimed to prioritize the local traffic. They choose the replica with the smallest value of this metric.

Web++ [74] is a middleware, that replicates dynamically and transparently Web applications on hosts located all over the world, in order to improve the applications' responsiveness and availability. Their replica selection approach also minimizes the response time expected to be perceived by the client. The response time is estimated by the metric called *S-percentile*. The formula of *S-percentile* combines the average and the variance of previously observed response time values. When computing the average, recent measures are weighted more heavily than old ones. The response time measures, observed passively, are stored within a latency table, maintained by the clients. In this table, the measures are associated a TTL parameter (in the order of minutes). When the response time measure for a given replica becomes obsolete, the client updates it by sending to that replica an asynchronous HEAD request. Their approach proved to improve the response time of Web applications by 36% on average, compared to the centralized case.

Leganet [22] is a lazy multi-master replication system aimed to improve the response time of transactions executed on a database cluster, while guaranteeing the required freshness constraints. They redirect the queries to secondary nodes by using a cost function, which takes into account the current load of the nodes and the freshness requirements of that query. A first approach to compute this cost function uses the value 1 as the cost of executing a transaction. The cost function becomes the sum between the load of a replica (measured as the number of transactions executing concurrently) and the number of refresh transactions which need to be propagated from the master before executing the query, so as to satisfy its freshness requirements.

A second approach, consider the cost function as an approximation of the transaction response time. In order to estimate transaction response time, they decompose the response time into the service time (obtained by running the transaction when the resources aren't shared) and the waiting time (due to resource contention). The waiting time is estimated by assuming that the resources (CPU and I/O) are equally utilized by transactions processed concurrently. The load of a node, monitored by a Unix tool, approximates the number of concurrent transactions. The waiting time increases proportionally with the number of concurrent transactions and to the service times of those transactions (compared to the service time of the transaction for which the response time is estimated).

A replica is considered as appropriate for a query transaction if it has the smallest cost function value. A replica is appropriate for an update transaction if its cost function value is inferior to a pre-defined threshold.

5.5 Systems providing flexible selection criteria

The anycast service [77] is a scalable replication middleware for Web services. The main innovation of this work consists in providing flexible replica selection policies. In this respect, they allow the users to specify within *filters*, the policies and/or the performance objective, stating what is a suitable replica. Examples of metrics which can be used within the filters include: server load, round-trip time, response time, as approximated by the transfer time of a small-sized probe file. The replica selection is performed by resolvers, organized in a DNS hierarchy. They apply the criteria contained in the filter, in descending order by their priorities and return a subset of replicas to the clients. The same criteria of the filter are applied client-side, in order to select one replica to which the client requests will be redirected.

They also proposed a response time estimator, for Web servers. This estimator is defined by a formula which multiplies two factors, counting for the server load, respectively for the network path characteristics. The two factors are obtained by measuring passively the service times of Web requests and by

system	system load	network proximity	response time estimator
PredictedTT [13]	-	-	x
Web++ [74]	-	-	x
Radar [53]	x	x	-
Weighted Total Response Time [9]	-	-	x
Globule [50]	-	x	-
anycast service [77]	x	x	x
Akamai [15]	x	-	-
AQuA [33]	-	-	x
Refresco [45]	-	-	x
Leganet [22]	-	-	x

Table 5.1: Metrics used for replica selection

collecting the transfer time of a well-known file (which contains the service times and dummy data). The service times are also pushed pro-actively by the Web servers to the redirectors. Their approach improves response time by 4 times, compared to a random selection and 2 times, compared to the number of hops metric.

AQuA [33] is a primary-backup replication system for distributed services, represented as CORBA objects. AQuA allows the users to define the QoS guarantees they need, with respect to the response time and to the replica consistency. The QoS specification, attached to a read-only transaction, contains the maximal threshold on response time, the minimal probability with which this threshold should be satisfied, and the maximal tolerated staleness. In order to satisfy the response time requirement, AQuA uses a probabilistic approach, wherein they predict that the response time, expected from a subset of replicas, satisfies the given threshold. The prediction approach is based on previously performed measurements of service times, measurements of response times of replies and on the current state of replicas with respect to the staleness threshold. The service time measures, observed passively by the replicas, are sent to all clients. They approximate the backup replicas staleness (defined as the number of missed updates), based on the time elapsed since the last propagation and the request inter-arrival frequency. Their probabilistic model determines the minimum number of replicas that are needed in order to guarantee the response time threshold, with the required probability. They guarantee the response time requirement, despite the failure of a member of the replica set.

Akamai [15] is a CDN, that targets improved availability of the replicated data and load-balancing of replica servers (so as to avoid flash crowds). They use a DNS-based content server selection mechanism, that optimizes one of the following metrics: CPU utilization, disk I/O utilization, network utilization. Their technique proved to be efficient when handling flash crowds.

Table 5.1 summarizes the approaches used by existing replica selection systems so as to find the suitable replica.

5.6 Conclusion

Existing approaches for response time optimization rely on static or dynamic metrics, that correlate with the response time. Static metrics include the geographical distance [27], the number of router hops [11, 16, 26, 59, 74] and the number of AS hops [49].

We classify the dynamic metrics into basic and composite metrics. The former are obtained by direct measurements, while the latter are computed by combinations of basic metrics. The basic dynamic metrics include the network latency (given by round trip time, for example), as used in [13, 11, 26, 59, 74], the available bandwidth, as used in [7], in Web Server Director [55], in [9, 15, 16], the HTTP request latency [59, 74], the transfer time of a given (probe) file [77] or server load [6, 15, 77]. Web++ [74] found the correlation between HTTP request latency and response time equal to 0.73.

The composite dynamic metrics include the metric used in anycast service [77], the metric used in Radar [53], *S-percentile* [74], *PredictedTT* [13], *Weighted Total Response Time* [9] and the metric defined in Leganet [22].

With respect to the state of art, we investigate more deeply the impact of the request workload and of the resource utilization on the response time expected to be obtained for that request.

Chapter 6

A Workload-Aware Response Time Estimator

6.1 Introduction

The *response time*, perceived by a client for its request to a given service, has been already accepted as the most relevant metric that correlates with the Quality of Service expected by the client from his/her replica. However, the response time is a composite metric, obtained after the execution of that request. In order to redirect the client requests to the appropriate replicas, replication management systems need to determine before executing the requests, which replica could deliver the best response time or a response time satisfying a given bound. The issue that we address in this chapter is how to estimate both accurately and efficiently the response time expected by the client for the requests he/she submits to his/her replica.

Existing replication systems rely on metrics, such as: the number of router hops, the round-trip time, the transfer time of a probe file or the bandwidth. Each of these metrics correlates with the response time for particular resource demands. For example, the round-trip time metric is appropriate for requests with small-sized replies. The bandwidth becomes the prevalent resource, in the case when the replies contain data of large size. We propose a response time estimator which takes into account the real resource demands of each request to the service. The estimator assigns weights to the processing capacity and to the current utilization of those resources available on a particular replica host.

The rest of this chapter is structured in nine sections. Section 6.2 depicts the independent components of the response time, and some notions used within estimations. Sections 6.3, 6.4, 6.5 and 6.6 detail the estimation of each individual component. Section 6.7 evaluates experimentally the accuracy of our estimation algorithm. Section 6.8 presents the metrology building brick, which provides the resource utilization measures, needed to perform the response time estimation. Section 6.9 presents the Response Time Estimator component, which implements the response time estimation algorithms. Finally, section 6.10 concludes the chapter.

6.2 Decomposition of response time

We derive our approach for response time estimation in two steps. Firstly, we decompose the response time, expected to be obtained by executing a service request, into independent components: the CPU service time, the disk I/O service time, the network transfer time, the CPU waiting time and the disk I/O waiting time. The service times correspond to the times intervals during which the CPU and/or the disk I/O resources are used by the request. The network transfer time counts for the time needed to transmit the request and the reply through the network. The CPU waiting time counts for the time period that the request spends in the ready queue (waiting for the CPU to become available). The disk I/O waiting time counts for the time period that the request spends in the disk I/O queue. We ignored from the decomposition, other components of response time, including: the client-side processing delay and the TCP/IP overhead. We assume that they are insignificant, with respect to the overall delay induced by the processing of the workloads. Overall, the estimation of the response time is obtained by estimating individual components, and summing these estimations.

Secondly, we estimate separately each component of the response time, thanks to a function that takes into account both the workload generated by the request and the processing capacity and the current status of the resources required to serve the request. Within this function, the resources that dominate the response time are weighted more heavily than the resources that the service uses lightly. The main benefit of our estimation approach is that it works for requests with heterogenous demands for various resources.

We denote the capacity and the utilization of the *host* of a particular replica by c, respectively, by u. The host capacity c has three components, representing the CPU capacity, the disk I/O capacity and the network connection capacity. These components are noted respectively by $c.cpu$, $c.io$ and $c.net$. The capacity is a static metric, that compares the processing performance expected from resources with the same utilization degree.

The host utilization u has three components, representing the CPU utilization, the disk I/O utilization and the network bandwidth utilization. These components are noted by $u.cpu$, $u.io$ and $u.net$.

The workload of a request quantifies the demands of that request for the three resources: CPU, disk I/O and network bandwidth. A request workload, noted wk, has three components, representing the demands of that request for the CPU, for the disk I/O and for the network bandwidth. They are noted respectively by $wk.cpu$, $wk.io$ and $wk.net$.

The response time estimator, applied for a given workload, is the sum of the service times, the network transfer time and the waiting times. Each of these components is parameterized by the workload of the request, and by the capacity and the current utilization of the resources of a particular host.

6.3 Estimating the CPU waiting time

We perform a regression-based analysis of the waiting time, where we study the dependency between the waiting times and the current utilization of the resources needed by the workload. Basically, each workload has its waiting time fitting curve, with the baseline points determined by measurements, or by estimations obtained from the measurements of other workloads.

Our experiments show that the contribution of the waiting time within the global response time becomes significant, especially under the conditions of medium to high resource utilization, where it increases exponentially. Computing the CPU waiting time is very challenging. The main difficulty arises from the fact that the waiting time depends on several parameters, such as: the request workload, the CPU processing capacity, the CPU utilization and the scheduling policies of the operating system. The big issue is to determine the contributions of each parameter to the overall CPU waiting time, and how the parameters interfere with each other. Another issue is to capture the dynamic values of the CPU utilization. In the following, we will define a function, noted by $cpuWt(wk, u)$, that estimates the CPU waiting time for the workload wk, under the host utilization u.

6.3.1 Empirical study

In order to gain some insights on the variation of the CPU waiting time, we performed two series of empirical studies, where we measured the real CPU waiting time. In both studies, we fixed the host capacity. In the former study, we fixed the workload and we varied successively the CPU and the disk I/O utilization. We represented graphically the evolution of the CPU waiting time according to the CPU utilization (and fixed disk I/O utilization values). In the latter study, we fixed both CPU and disk I/O utilization, and we varied successively the CPU workload and then the disk I/O workload. We represented graphically the evolution of the CPU waiting time, according to the CPU workload, respectively to the disk I/O workload. The experiments have been performed on machines with a Pentium 4 processor with 3GHz, 900MB RAM and running Linux.

6.3.1.1 Varying utilization

We measured the CPU waiting time for different workloads, while varying the CPU and the disk I/O utilization on the replica host.

Figure 6.1 shows six graphics for six workloads, where we neglected the network workload and we varied the ratio between the CPU and the disk I/O workload. We fixed the disk I/O utilization value to 0. We plotted on x-axis 20 values of the CPU utilization, and on y-axis the corresponding values measured for the CPU waiting time. All the six curves, obtained by unifying the points, follows an exponential evolution, growing more rapidly in the case of large CPU workloads. This result is valid for any value of the disk I/O utilization. Our

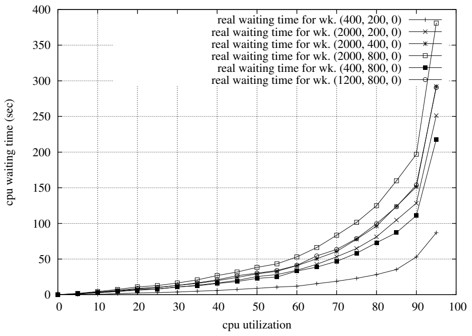

Figure 6.1: Measured CPU waiting time

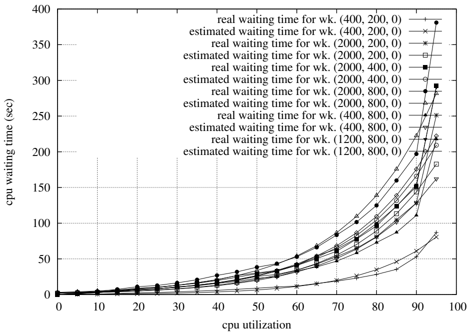

Figure 6.2: Estimated vs. measured CPU waiting time

experiments lead us to the conclusion that, generally, the CPU waiting time, measured for a given CPU workload, grows exponentially, as the CPU utilization increases.

With these elements, the function $cpuWt$ becomes an exponential function with the exponent expressed as a linear function. This function is parameterized by $u.cpu$ and has the form $y = a{*}u.cpu + b$, where the coefficients a and b depend both on the workload wk and on the utilization value u. We obtain the following formula for the waiting time:
$cpuWt(wk, u) = exp(a * u.cpu + b)$, where $a = f(wk, u)$ and $b = g(wk, u)$ (1)

In order to compute the coefficients a and b, we need the values of the CPU waiting time under at least two utilization values u_i with the same disk I/O utilization, i.e. $u_i.io = u_j.io$ and $u_i.cpu \neq u_j.cpu$, $i,\ j = 1, n,\ i \neq j,\ n \geq 2$. We note by m_i the CPU waiting times measured under u_i, $i = 1,\ n$, for wk, i.e. $m_i = measured\_cpuWt(wk, u_i)$, $i = 1,\ n$.

At this point, we compute the coefficients a and b by applying the Least Squares Line Fitting method [36]. Basically, this method determines a and b by minimizing the square error, that is $\sum_{i=1}^{n}(m_i - a * u_i.cpu - b)^2$. We implemented this method within the function $getCoef()$ in Figure 6.5. This is parameterized by n pairs $(x_i, y_i),\ i = 1,\ n$, that serve as the baseline points for the regression-based approximation. With m_i representing the waiting time obtained under the CPU utilization value x_i, we have $y_i = log(m_i)$.

We define in Figure 6.5 the base primitive $cpuWt\_base$ of our estimation approach. It contains the core of the regression-based waiting time approximation. It determines the waiting time for the workload wk under utilization u, by applying formula (1), where the coefficients a and b are computed using the function $getCoef()$.

In order to validate our approach, we took the same workloads from Figure 6.1, for which we compared the CPU waiting time computed by the function $cpuWt\_base$ with the real CPU waiting time, obtained by executing the workloads. Figure 6.2 shows these results, where the estimation curves are very close

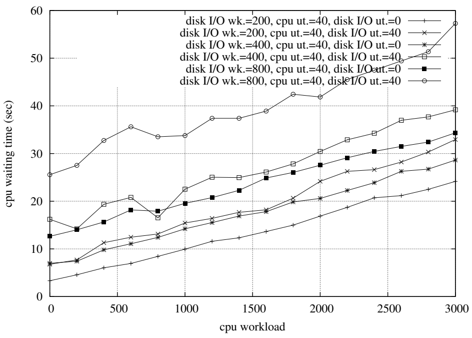

Figure 6.3: CPU waiting time variation with CPU workload

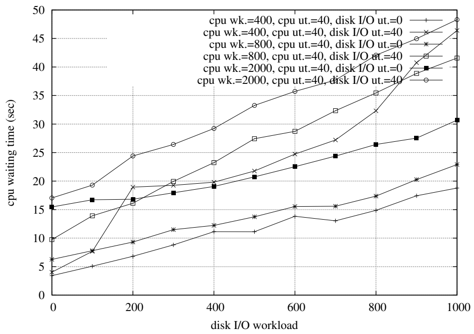

Figure 6.4: CPU waiting time variation with disk I/O workload

to the real measurements curves. Also, the estimated CPU waiting time correlates with the real CPU waiting time.

6.3.1.2 Varying CPU workload or disk I/O workload

We performed a second type of measurements, where we studied the evolution of the waiting time when the CPU workload, respectively, the disk I/O workload varies, while both CPU utilization and disk I/O utilization are fixed.

The graphs in Figure 6.3 show that the CPU waiting time varies linearly with the CPU workload. Precisely it increases linearly, as the values of the CPU workload increase. In all the graphs, we fixed the disk I/O workload, the CPU utilization and the disk I/O utilization.

The experiments we performed leads us to the conclusion that, under a given utilization value and fixed disk I/O workload, the CPU waiting time increases linearly with the CPU workload. Similarly, under a given utilization value and fixed CPU workload, the CPU waiting time also increases linearly with the disk I/O workload. This result is shown in Figure 6.4.

With variable CPU workload, respectively, disk I/O workload, we obtain the following formulas for the CPU waiting time:
$cpuWt(wk, u) = \alpha' * wk.cpu + \beta'$ (2) and
$cpuWt(wk, u) = \alpha'' * wk.io + \beta''$ (3)

In order to compute α' and β', we need the CPU waiting times under u, for several (at least two) workloads with the same disk I/O component as wk. We note them by wk_i, such that $wk_i.io = wk.io$, $wk_i \neq wk_j$ and the corresponding waiting times by m_i, $i,\ j = 1,\ k$, $i \neq j$, $k\ \geq\ 2$.

We apply the Least Squares Line Fitting method. Precisely, we compute α' and β' using the function $getCoef()$, parameterized by $x = \{wk_i.cpu\}$ and $y = \{m_i\}$, $i = 1,\ k$.

Similarly, in order to compute α'' and β'', we need the CPU waiting times under u, for several (at least two) workloads with the same CPU component as wk. We note them by wk_i, such that $wk_i.cpu = wk.cpu$, $wk_i \neq wk_j$, and the corresponding waiting times by m_i, $i, j = 1, k, i \neq j, k \geq 2$. We compute α'' and β'' by applying the function $getCoef()$, parameterized by $x = \{wk_i.io\}$ and $y = \{m_i\}$, $i = 1, k$.

6.3.2 Configuration data

The formulas (1), (2) and (3) serve as basis for estimating the CPU waiting time, for any workload, under any utilization value. In order to apply these formulas (possibly in various combinations), we need the measurements of the CPU waiting times for some workloads; under some utilization values. Basically, we need to define the configuration data for the formulas (1), (2) and (3), so as to perform the estimation of CPU waiting time for any workload, under any utilization value. These configuration data contain reference utilization points and reference workloads, that are defined in the next two paragraphs.

Reference points Formula (1) needs to be configured with the values of the CPU waiting time, measured under several utilization values (for a particular workload). We call these utilization values *reference points*. In the case of the CPU waiting time, there are at least two reference points with the same value for the disk I/O utilization. We formalize the set of reference points, noted P', as follows: $P' = \{u'_i\}$, $i = 1, n'$; $n' \geq 2$, where $\exists\, i, j, 1 \leq i, j \leq n'$, such that $u'_i.io = u'_j.io$ and $u'_i.cpu \neq u'_j.cpu$

Reference workloads Formulas (2) and (3) need to be configured with the values of the CPU waiting time, measured for several workloads, under a given utilization value. We call *reference workloads*, the workloads for which the CPU waiting time has been measured under any reference point. The set of reference workloads, noted W, satisfies the following property. There are four different reference workloads, wherein two couples of workloads have the same CPU workload, and there are other four different workloads, wherein two couples of workloads have the same disk I/O workload. The rationale behind this property will be explained later, when describing the estimation algorithms. More formally:
$W = \{wk_i\}$, $i = 1, m, m \geq 4$, where $\exists\, i, j, k\ and\ l, 1 \leq i, j, k, l \leq m$, such that
$wk_i.cpu = wk_k.cpu$, $wk_j.cpu = wk_l.cpu$, $wk_i \neq wk_j$ and
$\exists\, u, v, w\ and\ z, 1 \leq u, v, w, z \leq m$, such that
$wk_u.io = wk_w.io$, $wk_v.io = wk_z.io$, $wk_u \neq wk_v$.

We associate to each $wk \in W$, the set of the CPU waiting time measures, obtained under each reference point from P'. More formally:
$M'_{wk} = \{(u'_i, m'_i)\}$, where $m'_i = measured\_cpuWt(wk, u'_i)$, $u'_i \in P'$, $i = 1, n'$. We aggregate all the measures into the set M', where $M' = \cup_{wk \in W} M'_{wk}$. The sets P', W and M' are defined for every *host class* (i.e. machines with the same resource capacity, as defined in section 1.4).

6.3.3 Algorithms for CPU waiting time estimation

In this section, we show how to estimate the CPU waiting time, for any workload wk, under any utilization value u of the *host*. If wk isn't a reference workload, we distinguish two cases (1) wk has at least the CPU or the disk I/O component in common with a reference workload and (2) neither the CPU component, nor the disk I/O component of wk is contained by the reference workloads. In the former case, we name wk a *semi-known workload*, otherwise we name it an *unknown workload*. The following three sections show how the estimation proceeds if wk is a *reference workload*, a *semi-known workload* or an *unknown workload*. For each type of workload, we distinguish between the cases where u is a reference point or not.

6.3.3.1 The case of reference workloads

The function $cpuWt\_known(wk, u)$ (shown in Figure 6.5) estimates the response time for wk, considered as a reference workload. If u is a reference point from P', we obtain directly from M_{wk}, the CPU waiting time measured under u.

If u is not a reference point, in order to compute the coefficients a and b under the utilization value u, we choose from the set M_{wk}, n pairs $(u_i,\ m_i)$, such that $u_i.io = u_j.io$, with their common value $u_i.io$ being the closest to $u.io$, among all the reference points. With these elements, we compute $cpuWt(wk, u)$, by applying the function $cpuWt\_base()$, parameterized by the baseline points $(\{u_i.cpu\},\ \{m_i\})$, $i = 1,\ n$.

6.3.3.2 The case of semi-known workloads

The function $cpuWt\_semiknown(wk, u)$ (shown in Figure 6.5) estimates the response time for wk, considered as a semi-known workload. We consider that wk has the same disk I/O component as some reference workloads.

We consider k reference workloads $wk_i, i = 1, k$, that have the same disk I/O workload as wk. 1) If u is a reference point, we get from the sets M_{wk_i}, the CPU waiting time values m_i, measured under u for each workload wk_i. We compute $cpuWt(wk, u)$ by applying the formula (2).

If u isn't a reference point, we get from P', n reference points u_i, with the same disk I/O component, and compute $m_i = cpuWt(wk, u_i)$, using the previous case 1). Finally, we compute $cpuWt(wk, u)$ by applying the function $cpuWt\_base$, parameterized by the baseline points $(\{u_i.cpu\},\ \{m_i\})$, $i = 1,\ n$.

6.3.3.3 The case of unknown workloads

The function $cpuWt\_unknown(wk, u)$ (shown in Figure 6.5) estimates the response time for wk, considered as an unknown workload wk. We consider l sets, each one containing k reference workloads wk^i_u, such that: $wk^i_u.cpu = wk^i_v.cpu$, $i = 1,\ l;\ u, v = 1,\ k$. Let $wk^i.cpu = wk^i_u.cpu$, $wk'_i = (wk^i.cpu, wk.io)$, $i = 1,\ l$.

1) If u is a reference point, the algorithm proceeds in two steps. In the first step, we estimate the CPU waiting time m_i for the semi-known workload wk'_i, by applying the previous primitive $cpuWt\_semiknown()$.

In the second step it estimates the CPU waiting time, noted d, for the workload wk, by applying the regression formula (2), with the baseline points $(\{wk'_i.cpu\}, \{m_i\}), i = 1, l$.

2) If u is not a reference point, we get n reference points $u_i \in P$, with the same disk I/O component, i.e. $u_i \neq u_j, u_i.io = u_j.io$. We compute $m_i = cpuWt(wk, u_i), i = 1, n$ using the previous case 1). Finally, we estimate the CPU waiting time d for wk, by applying the function $cpuWt\_base()$, parameterized by the baseline points $x = \{u_i.cpu\}$ and $y = \{m_i\}, i = 1, n$.

6.4 Estimating the disk I/O waiting time

We note $diskIoWt(wk, u)$, the function estimating the disk I/O waiting time for the workload wk under u. We followed a similar approach, as in the case of the CPU waiting time. We performed the same two series of measurements. Firstly, we measured the disk I/O waiting time, while varying the value of the disk I/O utilization. The curves obtained follow this time a linear evolution, that we considered when deriving the formula transposing (1) to disk I/O waiting time. We obtain the following formula for the disk I/O waiting time:
$diskIoWt(wk, u) = a * u.io + b$ (4)

In order to compute the coefficients a and b, we need the measures of the disk I/O waiting time, under utilization values, with the same CPU component. We consider n points $u_i, i = 1, n$ such that $u_i.cpu = u_j.cpu$ and $u_i.io \neq u_j.io$ and m_i is the measure of the disk I/O waiting time under $u_i, i = 1, n$. In this case, a and b are computed using the function $getCoef()$, parameterized by $x = \{u_i.io\}$ and $y = \{m_i\}$, where $m_i = measured\_diskIoWt(u_i)$.

The second type of measurements showed that the disk I/O waiting time varies linearly with the workload. We obtain the following formulas similar to (2) and (3):
$diskIoWt(wk, u) = \alpha' * wk.io + \beta'$ (5) and
$diskIoWt(wk, u) = \alpha'' * wk.cpu + \beta''$ (6)

The coefficients α' and β' are computed using reference workloads with the same CPU component. We consider k reference workloads $wk_i \in W$ with the same CPU component, and m_i the value of the disk I/O waiting time measured for wk_i under $u_i, i = 1, k$. The coefficients α' and β' are computed by applying the function $getCoef()$, parameterized by $x = \{wk_i.io\}$ and $y = \{m_i\}$.
The coefficients α'' and β'' are computed, similarly, using reference workloads with the same disk I/O component.

The configuration data, needed in order to compute the coefficients, consist in the reference points P'', the reference workloads W and the associated

```
getCoef(n, x, y) {
  sx = Σ_{i=1}^{n} x_i; sy = Σ_{i=1}^{n} y_i; ssx = Σ_{i=1}^{n} x_i^2; sxy = Σ_{i=1}^{n} x_i * y_i;
  c = (n * ssx − sx^2); a = (n * sxy − sx * sy)/c; b = (sy * ssx − sx * sxy)/c;
  return < a, b >
}
cpuWt_base(wk, u, n, x, y){
  < a, b >= getCoef(n, x, y)
  d = exp(a * u.cpu + b)
  return d
}
cpuWt_known(wk, u){
  if (u ∈ P')
    m = measured_cpuWt(wk, u)
  else {
    a) get n pairs (u_i, m_i) from M'_wk, such that u_i ≠ u_j, u_i.io = u_j.io and |u_i.io − u.io| is minimal, i,
j = 1, n, i ≠ j
    b) let x = {u_i.cpu} and y = {log(m_i)}; d = cpuWt_base(wk, u, n, x, y)
  }
  return d
}
cpuWt_semiknown(wk, u){
  //wk has the same disk I/O workload as some reference workloads
  get wk_i ∈ W, wk.io = wk_i.io, i = 1, k
  if (u ∈ P') {
    m_i = measured_cpuWt(wk_i, u),
    a) let x = {w_i.cpu} and y = {m_i}; < α, β >= getCoef(n, x, y)
    b) d = α * wk.cpu + β
  } else {
    a) get n points u_i ∈ P', s.t. u_i ≠ u_j, u_i.io = u_j.io and |u_i.io − u.io| is minimal, i = 1, n
    b) m_i = cpuWt_semiknown(wk, u_i)
    let x = {u_i.cpu} and y = {log(m_i)}; d = cpuWt_base(wk, u, nx, y)
  }
  return d
}
cpuWt_unknown(wk, u){
  get l sets, each one containing k workloads wk_u^i ∈ W, wk_u^i.cpu = wk_v^i.cpu, i = 1, l; u, v = 1, k
  if (u ∈ P') {
    let wk^i.cpu = wk_u^i.cpu, wk'_i = (wk^i.cpu, wk.io), i = 1, l
    a) m_i = cpuWt_semiknown(wk'_i, u)
    b) let x = {wk'_i.cpu}andy = {m_i}; < α, β >= getCoef(n, x, y)
    d = α * wk.cpu + β
  } else {
    a) get n points u_i ∈ P', s.t. u_i ≠ u_j, u_i.io = u_j.io and |u_i.io − u.io| is minimal, i, j = 1, n, i ≠ j
    b) m_i = cpuWt_unknown(wk, u_i)
    let x = {u_i.cpu} and y = {log(m_i)}; i, j = 1, n, d = cpuWt_base(wk, u, n, x, y)
  }
  return d
}
cpuWt(wk, u) {
  if (wk is a reference workload)
   d = cpuWt_known(wk, u)
  else
    if (wk is a semi-known workload)
     d = cpuWt_semiknown(wk, u)
    else
     d = cpuWt_unknown(wk, u)
  return d
}
```

Figure 6.5: The estimation of the CPU waiting time

measures M'', defined in the next subsection.

6.4.1 Configuration data

In the case of the disk I/O waiting time, the set with the *reference points* contains two different reference points with the same value for the CPU utilization. More formally:
$P'' = \{u''_i\}$, $i = 1, n''$; $n'' \geq 2$, where $\exists\ i, j,\ 1 \leq i, j \leq n''$, such that $u''_i.cpu = u''_j.cpu$, $u''_i.io \neq u''_j.io$.

We consider the same set of *reference workloads*, as defined in section 6.3.2. We measured the disk I/O waiting time, under each workload $wk \in W$ and each reference point $u \in P''$.
We obtain $M''_{wk} = (u''_i, m''_i)$, where $m''_i = measured\_diskIoWt(wk, u''_i)$,
$u'' \in P''$, $i = 1, n''$. We note $M'' = \cup_{wk \in W} M''_{wk}$. The sets P'' and M'' are defined for every *host class*.

6.4.2 Algorithms for disk I/O waiting time estimation

Similarly to CPU waiting time, we defined the base function $diskIoWt\_base(wk, u)$. We performed the estimation of the disk I/O waiting time, by distinguishing between *reference workloads*, *semi-known workloads* and *unknown workloads*, within the functions $diskIoWt\_known(wk, u)$, $diskIoWt\_semiknown(wk, u)$, respectively $diskIoWt\_unknown(wk, u)$, presented in Figure 6.6.

6.5 Estimating the service times

The *CPU service time* is estimated by dividing the CPU workload by the CPU capacity (expressed as the number of MFLOPS per second). The *disk I/O service time* is estimated by dividing the disk I/O workload by the disk I/O capacity (expressed in MB/s).

6.6 Estimating the network transfer time

The *network transfer time* is also estimated by means of a regression based technique. We distinguish between bandwidth-intensive vs. non-bandwidth intensive network workloads. The former workloads have the transfer time dominated by the bandwidth, while the latter workloads have the transfer time dominated by the round-trip-time. We discuss here only the transfer time for the bandwidth-intensive workloads. A workload is considered as bandwidth intensive if its value is greater than the maximal bandwidth between a host and a client. We approximate the bandwidth between a host and a client by the bandwidth of the link connecting the host to the closest network router. The rationale behind this chaise is two-fold. On one hand, it is very difficult to have accurate measures of the available bandwidth between any replica host and any client, without a considerable overhead with respect to the network traffic generated by such measurements. On the other hand, if the network

```
diskIoWt_base(wk, u, n, x, y){
  < a, b >= getCoef(n, x, y)
  d = a * u.io + b
  return d
}
diskIoWt_known(wk, u){
  if (u ∈ P'')
    d = measured_diskIoWt(wk, u)
  else {
    get n pairs (u_i, m_i) fromM''_wk, such that u_i ≠ u_j u_i.cpu = u_j.cpu, and |u_i.cpu − u.cpu| is minimal,
i, j = 1, n, i ≠ j
    let x = {u_i.io}, y = {m_i}, d = diskIoWt_base(wk, u, n, x, y)
  }
  return d
}
diskIoWt_semiknown(wk, u) {
  //wk has the same disk I/O workload as two reference workloads
  get wk_i ∈ W, wk.io = wk_i.io, i = 1, k
  if (u ∈ P'') {
    m_i = measured_diskIoWt(wk_i, u)
    let x = {wk.cpu}, y = {m_i}; < α, β >= getCoef(n, x, y)
    d = α * wk.cpu + β
  } else {
     get n reference points u_i ∈ P', s.t. u_i ≠ u_j, u_i.cpu = u_j.cpu and |u_i.cpu − u.cpu| is minimal;
i, j = 1, n
     m_i = diskIoWt_semiknown(wk, u_i)
     let x = {u_i.io}, y = {m_i}; d = diskIoWt_base(wk, u, n, x, y)
  }
  return d
}
diskIoWt_unknown(wk, u){
  get l sets, each one containing k workloads wk^i_u ∈ W, wk^i_u.cpu = wk^i_v.cpu, u; v = 1, k; u ≠ v; i = 1, l
  if (u ∈ P'') {
    let wk'_i = (wk^i_u.cpu, wk.io)
    m_i = diskIoWt_semiknown(wk'_i, u)
    let x = {wk'_i.cpu}, y = {m_i}; < α, β >= getCoef(n, x, y)
    d = α * wk.cpu + β
  } else {
    get n reference points u_i ∈ P'', s.t. u_i ≠ u_j, u_i.cpu = u_j.cpu and |u_i.cpu − u.cpu| is minimal;
i, j = 1, n
    m_i = diskIoWt_unknown(wk, u)
    let x = {u_i.io}, y = {m_i}, d = diskIoWt_base(wk, u, n, x, y)
  }
  return d
}
diskIoWt(wk, u) {
  if (wk is a reference workload)
    d = diskIoWt_known(wk, u)
  else
    if (wk is a semi-known workload)
      d = diskIoWt_semiknown(wk, u)
    else
     d = diskIoWt_unknown(wk, u)
  return d
}
```

Figure 6.6: The estimation of the disk I/O waiting time

congestion occurs on a link close to the client host, and the network bandwidth is the prevalent resource, no server-side selection algorithm could help, because all replicas will perform similarly bad.

In this case, the transfer time is given by the formula:
$nt(wk, host) = wk.net/(a * host.c.net + b)$,
where *host.c.net* is the network bandwidth of *host*. We compute a and b, for some known workloads, for which there are available some measures of transfer time, under some bandwidth utilization values. These measures are used as the baseline points within the Least Squares Line Fitting method, that gives a and b. For the other unknown workloads, the transfer time is computed using the transfer time of reference workloads under the same utilization value. This computation relies on the observation that the transfer time increases linearly with the network workload, under a given bandwidth utilization value.

6.6.1 The definition of the response time estimator metric

The response time estimator metric, noted *RT_estimator*, approximating the response time, expected to be obtained by running the workload *wk* on *host*, becomes the sum of the *CPU service time*, the *disk I/O service time*, the *CPU waiting time*, the *disk I/O waiting time* and the *network transfer time*. Figure 6.7 shows the mathematical formulas for *RT_estimator* and for its components.

$RT\_estimator(wk, host) = cpuSt(wk, host.c) + diskIoSt(wk, host.c) + cpuWt(wk, host.u) + diskIoWt(wk, host.u) + nt(wk, host)$
$cpuSt(wk, c) = wk.cpu/c.cpu$
$ioSt(wk, c) = wk.io/c.io$

Figure 6.7: Formalizing the estimation function

6.7 Experimental validation

6.7.1 The accuracy of the response time estimator

In order to evaluate the accuracy of the response time estimator, we performed experiments for various concrete workloads. We used a Workload Simulator, that simulates the execution of requests. For each request, it generates a *test application* with the same workload as the request. The test application makes use of matrix multiplication, generating the number of MFLOPS specified in the CPU workload, plus basic read/write disk I/O accesses equivalent to the total data size, specified in the disk I/O workload, plus read/write socket accesses equivalent to the total data size, specified in the network workload.

We used the reference points set $P' = P'' = \{u_i\}$, $u_i.cpu, u_i.io \in \{0, 20, 40, 80\}$, $i = 1,\ 3$. As the baseline measures, we took the CPU waiting time, under three CPU utilization values $\{20, 40, 80\}$ and the disk I/O waiting time under the same values considered for disk I/O utilization.

For a given workload wk, to be executed on a given $host$, the experimentation follows the following scenario. We vary the values of the CPU, disk I/O and network bandwidth utilization, available on that host. Under a given utilization value, i.e. the triple (CPU utilization, disk I/O utilization, network bandwidth utilization), we determine the couple containing the response time estimated with our approach and the real response time measured by executing the workload. For n utilization values, we obtain the set D with n measures of response time. Each sample i is defined as the couple (*estimated response time*, *real response time*), noted by ert_i, respectively rrt_i. More formally:
$D = \{(ert_i, rrt_i)\}$, $i = 1,\ n$ and $ert_i < ert_{i+1}$, $\forall\ i, i = 1,\ n-1$. We draw the graph associated to D, plotting on x-axis the values ert_i and on y-axis the corresponding values rrt_i.

We define the *value estimation error* for each sample in D. We note it by $valEr_i$, where $i = 1,\ n$, $valEr_i = min(100 * |ert_i - rrt_i|/rrt_i, 100)$

We define the *variation estimation error* for each two successive samples in D. We note it by $varEr_i$, where $i = 2,\ n$, $varEr_i = min(100 * (rrt_i - rrt_{i-1})/rrt_{i-1}, 100)$, if $rrt_i > rrt_{i-1}$, and 0 otherwise.

6.7.1.1 Experimenting workloads without network bandwidth demands

We begin the validation of our response time estimation approach, by considering only workloads, where the network component is 0. Specifically, we experimented our approach for three workloads (2000, 800, 0), (2000, 400, 0) and (400, 800, 0), varying the ratio between the computation demands vs. the disk I/O demands. For each workload, we considered both the cases where it was a *reference workload* (in which case the estimation uses its measures under the reference points), respectively *unknown* (in which case the estimation relies on four reference workloads). The reference workloads used were: (400, 200, 0), (400, 2000, 0), (3000, 200, 0) and (3000, 2000, 0).

The Figure 6.8 shows the results in the case of the workload (2000, 800, 0), Figure 6.9 for the workload (2000, 400, 0) and Figure 6.10 for the workload (400, 800, 0). We represented 44 points on each graph. Each point on a graph, corresponds to a particular CPU and disk I/O utilization value, and is represented by plotting on the x-axis the estimated response time and on y-axis the real measured response time (both of them obtained under that utilization value).

In each case, we compared the response time estimated by our approach with the real response time. One can see that in each graph, the curves corresponding to the estimated response time vs. the real response time are close to each other (being closer in the case of the reference workloads). Another important observation is that the variation of the real response time matches with good accuracy the variation of the estimated response time (i.e. the curve correlating the estimated response time with the real response time is mostly monotone increasing). The large variations at the end of the graphs are due to large values of the resource utilization (between 90%-100%), where the estimation isn't satisfactory. In fact, the accuracy of estimation decreases, as the

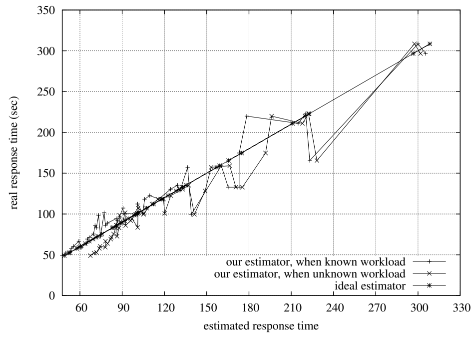

Figure 6.8: Estimation for wk. (2000, 800, 0)

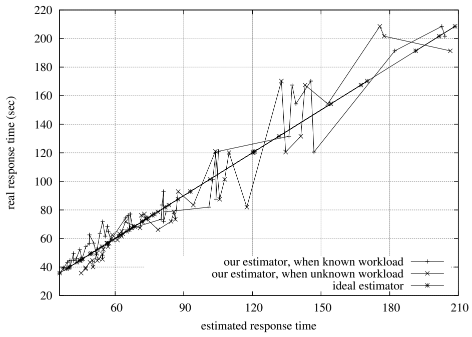

Figure 6.9: Estimation for wk. (2000, 400, 0)

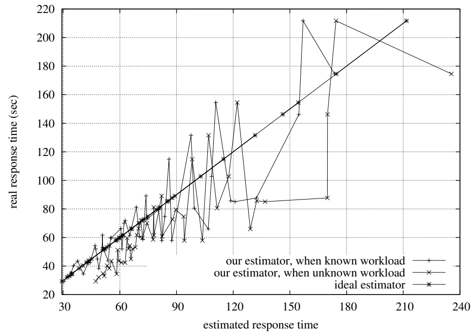

Figure 6.10: Estimation for wk. (400, 800, 0)

resource utilization increases.

6.7.1.2 Experimenting workloads with demands for all resources

In this section, we show the results obtained in the case of workloads with demands for all resources. The experimentation follows a scenario similar to the case of workloads without network bandwidth demands. However, in this case, we also vary the network bandwidth utilization, when determining the graph points. We represented 64 points on each graph.

The graphs in Figures 6.11 and 6.12 show the response time estimation for the workloads (2000, 800, 20) and (2000, 800, 80), considered successively reference workloads and unknown workloads. One can see that the monotony of the real response time matches, in most cases, that of the estimated response time (i.e. in most cases, a bigger estimated response time corresponds to a bigger real response time). Comparing the results of our estimator with an ideal estimator, one can see that there is an over-estimation of the response time, but this still remains under reasonable limits. We also have noticed that the unsatisfactory estimation occurs for large values of the utilization. This explains the large variations at the end of the graphs. When the values of

Workload	coef. a and b, when reference wk.	coef. a and b, when unknown wk.
(2000, 800, 0)	0.91 12.08	0.99 -8.45
(2000, 400, 0)	1.01 3.94	1.15 -13.77
(400, 800, 0)	0.94 2.46	0.87 -4.71
(2000, 800, 20)	1.00 10.74	1.12 -17.40
(2000, 800, 80)	0.56 55.44	0.61 39.90

Table 6.1: Coefficients a and b for various reference and unknown workloads

utilization are small or even reasonably high, the estimation works fine. As in the previous type of workloads, the accuracy of estimation decreases, as the resource utilization increases.

We adopted two means to validate our model: by performing a regression based analysis of the estimation results and by studying the cumulative distribution of the estimation errors. In the former evaluation scenario, we studied the linear relationship between the estimated response time and the real response time, using as the baseline a set of 44 points resulted from measurements. Precisely, we determined the coefficients a and b of the line $y = a * x + b$, where x represents the estimated response time and y is the real response time. We represented the results in Table 6.1. Each row corresponds to a workload, considered a reference workload vs. an unknown workload. The ideal values, corresponding to the perfect estimation, are $a = 1$ and $b = 0$. The values, obtained for the coefficients, are close to the ideal values, except for the last case, when the network workload is equally important as the CPU and disk I/O workload.

With respect to the second evaluation scenario, Figures 6.13 and 6.14 shows the cumulative distribution for the value estimation errors, considering the case of reference workloads, respectively unknown workloads. A point on the graph is defined by plotting on x-axis the value estimation error, and on y-axis the percentage of estimations whose error is inferior to the x-axis value. These results show that the estimation accuracy is satisfactory. Figures 6.15 and 6.16 shows the cumulative distribution for the variation estimation errors. A point on the graph is defined by plotting on x-axis the variation estimation error, and on y-axis the percentage of estimations whose error is inferior to the x-axis value. These results show a good correlation between the estimated response time and the real response time.

The main benefit of the proposed solution is its accuracy, for workloads with a single or two bottleneck resources (as showed by the correlation coefficients in Table 6.1). Furthermore, the experiments showed that the estimation function orders correctly the replicas compared to real response time measures.

6.7.2 Choosing the reference points

An important question is how to choose the reference points. We conducted some experiments in order to answer to this question. We varied the set of utilization reference points, both in terms of number and distribution over the range [0, 100]. For each reference points set, we estimated the response time for seven different workloads. For each workload, we computed the value es-

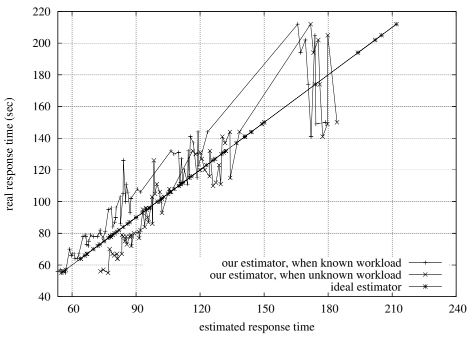

Figure 6.11: Estimation for wk. (2000, 800, 20)

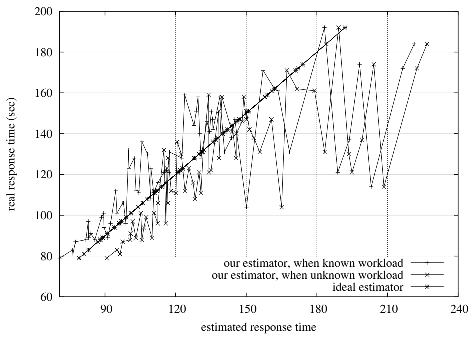

Figure 6.12: Estimation for wk. (2000, 800, 80)

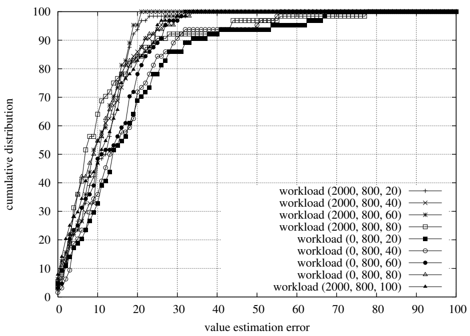

Figure 6.13: Value estimation error for reference workloads

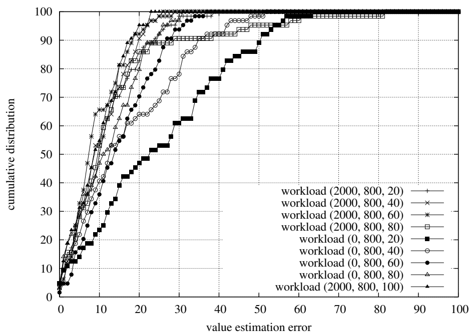

Figure 6.14: Value estimation error for unknown workloads

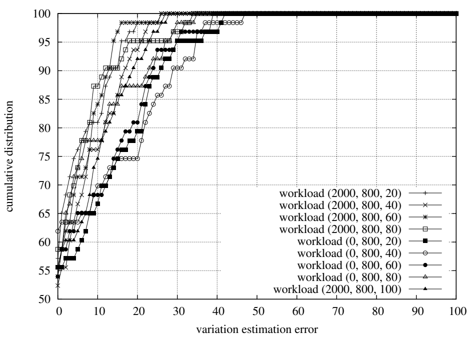

Figure 6.15: Variation estimation error for reference workloads

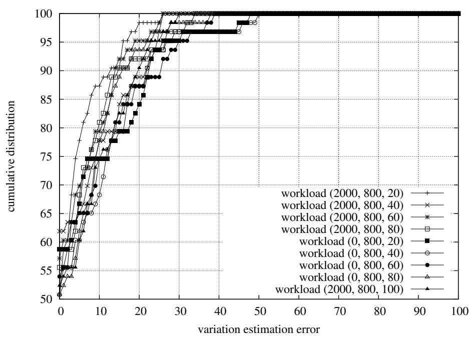

Figure 6.16: Variation estimation error for unknown workloads

Reference utiliz. points	Error for wk (400, 200, 0)	Error for wk (2000, 200, 0)	Error for wk (2000, 400, 0)	Error for wk (2000, 600, 0)	Error for wk (2000, 800, 0)	Error for wk (400, 800, 0)	Error for wk (1200, 800, 0)
10 20	241798.20	9078.83	2882.08	7205.66	12688.70	4694.38	10323.13
20 30	23.51	100.77	195.83	130.55	167.97	363.20	128.94
80 90	19.49	90.46	102.91	123.30	134.59	81.01	99.71
20 60	40.49	116.10	120.43	156.56	174.76	76.57	136.80
10 20 30	3482.33	941.28	854.83	1307.19	1306.20	1323.72	1033.57
30 50 70	36.15	101.70	133.55	144.66	148.14	103.53	118.01
20 40 60	38.13	112.93	114.00	149.02	165.76	73.64	131.06
10 40 80	17.50	71.97	85.61	93.93	105.83	61.38	75.25
10 15 20 25 30	2014.77	824.78	818.11	1097.96	1103.53	977.14	876.76
20 40 60 70 80	36.73	105.52	118.32	138.10	157.01	85.16	117.79
20 20 40 60 80	36.41	104.94	112.33	135.12	155.92	80.15	114.91
10 20 40 60 70	22.55	73.54	90.97	100.95	106.92	61.21	83.10
10 20 40 60 80	18.02	79.16	91.71	104.79	117.22	63.44	85.11

Table 6.2: Estimation errors when varying the set of reference points

timation error over a testing set with 20 CPU waiting time values. We show the results in Table 6.2. The main conclusion is that increasing the number of reference points doesn't improve the accuracy, unless the points are well chosen, i.e. uniformly distributed around "the middle' (i.e. the value 50%).

6.8 The Metrology System

In order to approximate accurately the response time expected from a set of replicas for a given request, the *response time estimator* needs the measures of the utilization of the resources, that are relevant for that request. In this respect, we developed a Metrology System, composed of Host Monitors and Metrology Servers. A Host Monitor is responsible for collecting the measures of CPU utilization, disk I/O utilization and system load. It sends the measures to a Metrology Server.

6.8.1 Metrology Server

The distribution of the Metrology Servers follows the grouping of hosts in *domains*. A Metrology Server instance is attached to one domain. It maintains for the hosts located in this domain, the history with the measures of the resource utilization metrics. Each measure is registered into the repository together with the timestamp of the instant when it was received. A Metrology Server stores and provides access to the measures and to derived samples, computed by applying various statistics over those measures. The statistics, that we consider, provide the following derived samples: the mean measure, the most recent measure, the most frequent measure, the median measure.

6.8.2 Host Monitor

There is a single Host Monitor instance on each replica host. Its role is to collect for the current machine, the dynamic measures of three main metrics: CPU utilization, disk I/O utilization and system load. The *CPU utilization* counts for the degree of the processor utilization (expressed in percents). The *disk I/O* *utilization* counts for the degree of the disk I/O utilization. Under Linux, we collect the measures of the CPU utilization metric by using the tool sar and the measures of the disk I/O utilization by using the tool iostat. We quantify the system load by the number of active processes in the system, at a given instant.

The Host Monitor object maintains a map associating to each of the three supported metrics an object, which defines how to aggregate a history of measures into a derived sample. This object contains the statistical computation, to be performed over the measures, and the time period, delimiting the measures to be considered within the computation. The Host Monitor collects periodically the measures for each metric, and use them in order to compute the value of the corresponding derived sample. The Host Monitor sends the derived sample values pro-actively to the Metrology Server, attached to the domain to which the current host belongs.

In our prototype implementation, we use the statistical computation type claiming for the most recent measure, for all the three metrics. We address two main issues: how often the metric measures should be collected (i.e. what is the optimal frequency of measurements)? and how often the Host Monitor should send the values of the derived samples to the Metrology Server (i.e. what is the dissemination criterion)? In the following three paragraphs, we firstly present the algorithm performed by the Host Monitor and then we answer to these issues.

The Host Monitor algorithm Our monitoring approach aims to achieve scalability, by localizing the traffic induced by the measures dissemination, at the scope of domains and avoiding to generate traffic on distant expensive network links.

The algorithm, performed by the Host Monitor, reiterates through the following three steps. Firstly, the Host Monitor sleeps the time period, common to all the derived samples. When the time period elapses, it wakes up and collects the metric measures. Finally, it decides if the measures should be sent to the Metrology Server or not, and proceeds to dissemination, if this is needed.

The measurement frequency We use the water-marking technique, by associating to each metric two thresholds, noted *underLoadThreshold* and *overLoadThreshold*. A metric measure inferior to *underLoadThreshold* indicates that the resource utilization is very low, so the resource can accept new services while still providing good performance. A metric measure superior to *overloadThreshold* indicates that the resource contention is already high. Consequently, allocating new services to that host should be prohibited, as it would conduct to dramatically decreased performance. A metric measure between *underLoadThreshold* and *overLoadThreshold* indicates that the resource is reasonably loaded. Consequently, it can still be used by new services.

We consider two monitoring time periods, noted *longPeriod* and *shortPeriod*. The measurement interval has the value *longPeriod*, when the measures are inferior to *underLoadThreshold* or superior to *overLoadThreshold*. In the former case, we use the observations of our experiments described in section 6.3.1, according to which when the resource is lightly used, small changes in its values have little impact on the response time that will be perceived by the client requests (as the waiting time curve hasn't begin to grow exponentially yet [19]). When the resource is overloaded by concurrent services, we also use *longPeriod* as the measurement interval, because the resource is unusable anyway and we don't want to increase the system load by the monitoring. When the measures are between *underLoadThreshold* and *overLoadThreshold*, we use the measurement interval *shortPeriod*, as in this case, any change in the measures could have a considerable impact on the response time.

The dissemination criterion The criterion for measures dissemination is defined so as to meet two complementary goals: maximizing the measures accuracy and minimizing the network traffic induced by the measures.

Precisely, our dissemination technique aims to capture significant variations between the measure, that has been previously sent to the Metrology Server and a newly collected measure. In other words, we ignore new measures, when they are close to the most recent disseminated measure.

We define the relation of *closeness* between two measures, by means of two configuration parameters: a function f and a threshold b. We say that two measures m_1 and m_2 are close if $|m_1 - m_2| < b$ (1).

If m_1 is the previous disseminated measure, and m_2 is the most recent collected measure and m_1 and m_2 are not *close*, (i.e. they don't satisfy the relation (1)), we switch to *shortPeriod*, as the time period preceding the next measurement. We note this new measure by m_3. If m_1 and m_3 are not close, the Host Monitor sends the measure m_3 to the Metrology Server. The monitoring interval is set according to the state denoted by m_3. We call *dissemination interval*, the time period to wait until the next dissemination, determined by the *closeness* relation. This period is bounded by the sum between *shortPeriod* and *longPeriod*.

6.8.3 Calibrating the measures dissemination

In our experiments, we consider a set of hosts running replicas of a service, with a given workload. We approximate the response time expected to be provided by each replica, by using the *response time estimator*, presented in chapter 6. We select the host, whose replica is expected to provide the best response time.

We studied experimentally the tradeoff between two metrics: the selection error, when looking for the host with the best estimated response time, and the overhead of collecting the measures used by the response time estimator. The selection error captures the impact of obsolete metric values on the response time estimation.

We consider all hosts ranked by their response times, estimated with real metric values. The *selection error* is defined as the position of the selected host in this ordered sequence, divided by the number of all hosts. The values of the *selection error* belongs to the interval (0, 1]. The value 1/n is obtained when the ideal host is selected. The value 1 is obtained when the worst host is selected. The selection error depends directly on the accuracy of the measures that the estimator uses. Precisely, it requires that the difference between the measures used and the real measures to be as little as possible. However, in order to limit this difference, the monitors should increase the frequency of the measurements and the frequency of the measures dissemination to the Metrology Servers, attached to their domains. Consequently, the system load, induced by measurements, and the network traffic, induced by the disseminations, increases. We define the *metrology overhead*, as the number of bytes exchanged between the Host Monitors and the Metrology Server, during a given time period.

We call *measure discrepancy*, the difference between the measure used by the *response time estimator* and the real measure. In our experiments, we looked for the optimal threshold on the *measure discrepancy*, defined for two metrics: CPU utilization and disk I/O utilization, so that to minimize selection error.

We simulated 289 replicas on the same machine, which is a Pentium 4 with 3GHz, 900MB RAM, running Linux. On this machine, we varied the CPU utilization, by running a background process, able to generate any CPU utilization value, given as parameter.

We considered several workloads, with the CPU workload varying between 0 and 2000 MFLOPS and the disk I/O workload varying between 0 and 800 MB. We varied the *measure discrepancy* between 10% and 60%.

We run the selection process for each configuration, including a given measure discrepancy and a given workload, and we computed the corresponding selection error. In this respect, the hosts were ranked according to their response time values estimated using the correct CPU and disk I/O utilization values.

For each measure discrepancy value, we computed the average selection error of all the workloads considered. The results, represented in Figure 6.17, show that the values of the selection error increase linearly with the measure discrepancy values. Also, the results in Table 6.3 show that even with a measure discrepancy value of 40%, our strategy selects a host among the first 50% of all ranked hosts (according to their estimated response time values).

In order to obtain the metrology overhead, for each value of the measure discrepancy, we computed the value of the network *bandwidth* consumed by the Host Monitors, in order to disseminate the measures. In each case, the Host Monitors collected 60000 measures of the resource utilization. The results, represented in Figure 6.18, show that the traffic generated increases exponentially with the measure discrepancy value.

In order to determine the appropriate measure discrepancy, we define an optimization function, parameterized with two successive discrepancy values t_1 and t_2. This function quantifies the degradation of the selection strategy when increasing the discrepancy value, in terms of decreased accuracy with respect the reduced bandwidth. We note the selection error corresponding to t_1 and t_2 by a_1, respectively by a_2. We also note the bandwidth consumed in t_1 and in t_2 by

Measure discrepancy	Selection error	Traffic generated	Optimization fct. value
10%	0.0959421	1185.33 KB	not def.
15%	0.141868	1054.99 KB	0.50348
20%	0.190311	863.951 KB	0.407918
25%	0.321799	700.503 KB	0.752804
30%	0.376848	552.971 KB	0.528213
35%	0.434099	442.198 KB	0.851602
40%	0.492293	355.458 KB	0.851131
45%	0.553004	286.455 KB	1.36711
50%	0.609626	232.254 KB	1.36208
55%	0.673168	175.255 KB	1.65502
60%	0.731362	146.557 KB	2.57255

Table 6.3: The tradeoff between the selection error vs. the metrology overhead

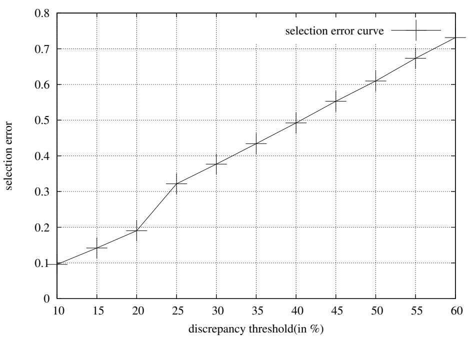

Figure 6.17: The selection error

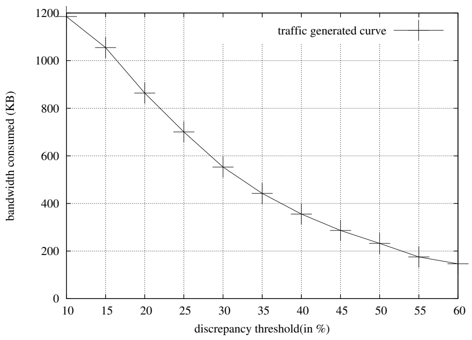

Figure 6.18: The metrology overhead

b_1, respectively by b_2. The minimization function, noted $f(t_1,\ t_2)$, is defined as $(a_2\ \text{-}\ a_1)/((b_1\ \text{-}\ b_2)/mawBw)$, where $maxBw$ is the bandwidth consumed for the minimum discrepancy value, which is equal to 10% in our experiments. We use the division by $maxBw$, so as to have the same value interval equal to (0, 1], for the two terms, which are divided in the formula of the function. Intuitively, we are looking for t_2, which do not decreases too much the selection error compared to the selection error in t_1, while reducing the metrology overhead, as much as possible.

We represented the minimization function in Table 6.3 and Figure 6.19. We choose the measure discrepancy, by setting a superior bound on the selection error and looking for the minimum function value, computed under the selection errors which satisfy that bound. For example, with the selection error set to 0.3, the measure discrepancy equal to 20% seems to be a good choice.

6.9 The Response Time Estimator component

We developed a component called Response Time Estimator, that implements the algorithms described in Figures 6.5, 6.6 and 6.7, in order to estimate the

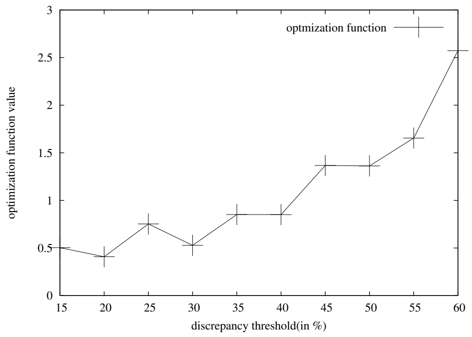

Figure 6.19: The optimization function

response time expected to be observed by executing a given workload, on a given host.

Each Response Time Estimator instance is configured with the CPU waiting time measures and the disk I/O waiting time measures, collected by running the reference workloads, on a set of representative hosts within the domain, under all the reference points of resource utilization. The measures are maintained as a map, indexed by the hosts (as shown in Figure 6.20). The data corresponding to a host contains the resource capacities and a map, indexed by the reference workloads. The data corresponding to a reference workload contains the CPU waiting time measures and the disk I/O waiting time measures obtained by running that reference workload on that host, under each reference point. We bound, by a predefined parameter, the number of hosts for which the measures are maintained in the memory. For all the hosts within the domain, the measures are stored on the disk, within configuration files. We configure each Metrology Server instance, attached to a given domain, with a Response Time Estimator instance.

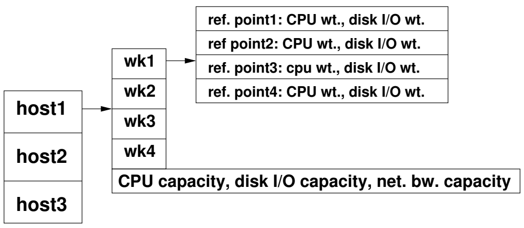

Figure 6.20: The configuration data of Response Time Estimator

6.10 Conclusion

We developed a response time estimator, capable to approximate the execution time expected to be obtained by running a given request on a given host. We showed experimentally that the estimated response time values correlate with the effective response time measured by executing the requests. Also, for requests with a given predominant resource, the estimated values are close to the measured ones. We also showed experimentally that the response time estimator remains reasonably accurate even if it uses measures that diverge from the most recent ones, provided that the discrepancy respects a given bound.

Chapter 7

A Generic and Customizable Replica Selection Protocol

7.1 Introduction

The replication decisions, already identified in existing replication systems [53], include: replica selection at the client connection-time, rebinding during client - replica interaction, replica creation, replica placement, replica deletion and replica migration. The replica selection decision selects the pertinent replica for a given client, among (a subset of) all available replicas. The rebinding decision determines when the replica, to which a given client is currently bound, becomes unsuitable and must be replaced by another replica. The creation decision determines when a new replica is needed. The placement decision selects the appropriate host, where the newly created replica should be placed. The migration decision decides when an existing replica must be moved on another host, because the host, where it is currently running, becomes overloaded. The deletion decision decides when an existing replica becomes unuseful and, consequently, it must be eliminated. Our work targets mainly the replica selection and the rebinding decisions.

The replication efficiency depends on the ability of the system to redirect the client requests to the appropriate replicas. The issue that we address in this chapter is how to find the appropriate replica for each client requiring access to a given replicated service. A subsequent issue, determined by our base hypothesis, is how to provide the various response time requirements of services with various resource demands by a single replica selection protocol. We developed a protocol performing the selection of the suitable replica, at the beginning of the session and, dynamically, when the current replica provides bad performance. The replica selection protocol aims to satisfy the performance requirements for a given percentage of requests. The protocol is configured with the replica selection and the rebinding criteria, both of them inferred from the response time requirements, specified by the service supplier. The replica selection criterion contains the conditions that must be fulfilled by a given replica, in order to be assigned to a given client. The rebinding criterion specifies how many times the violation of the performance requirements is tolerated, for a given replica accessed by a client, before selecting another replica for that client.

The protocol respects two main objectives: genericity and customizability to service-specific requirements. The need of providing these features is motivated by the heterogeneity of the services, with respect to their demands for various resources, such as the CPU, the disk I/O or the network bandwidth. The replica selection and the rebinding criteria vary from a service to another, according to their resources demands. However, we wish to support all possible criteria by a single protocol. Hence, the need to make the protocol generic. Also, the protocol should provide the performance requirements for a particular service, for which it is instantiated. Hence, the need to make the protocol customizable to service-specific criterion.

The rest of this chapter is structured in seven sections. Section 7.2 presents the specification of the performance requirements, as expected from the service supplier. Section 7.3 presents the concepts needed in order to provide the genericity feature of the replica selection protocol. The concepts include the derived metrics, the metric-based predicates and the abstract replica selection and rebinding criteria. Section 7.4 presents how the replica selection criterion is inferred, for each service, from its performance requirements. Section 7.5 presents how the rebinding criterion is inferred, for each service, from its performance requirements. Section 7.6 presents the Replica Manager component that implements the replica selection and the rebinding algorithms. Section 7.7 shows experimentally the accuracy and the scalability of the replica selection protocol and the benefits of re-selecting the pertinent replica at run-time. Finally, section 7.8 concludes the chapter.

7.2 Expressing the performance requirements

We use XML as the language support for specifying the requirements on the response time by the service-supplier. Figure 7.1 shows the DTD of a performance contract, which contains one or several performance requirements. Each requirement is associated to one or several operations of the service and is defined as a best-effort or as a relative predicate on the *response time*. The best-effort requirement claims for the best response time value, expected to be provided by the selected replica. The relative requirement contains a threshold, which defines what are the valid values of the *response time*. In both cases, the requirement also contains the average workload generated by the operations to which the requirement is attached.

```
<!ELEMENT performance_contract (requirement+)>
<!ATTLIST requirement operations CDATA>
<!ATTLIST requirement workload CDATA>
<!ATTLIST requirement workload_class CDATA>
<!ATTLIST bound CDATA>
```

Figure 7.1: The DTD of a performance contract

The definition of the workloads, attached to operations, is a challenging task for the service suppliers. Ideally, the workload characteristics should be computed automatically for each operation of the service, by a profiling tool.

In our work, if the service supplier doesn't know the exact workload generated by operations, he/she has the option to specify, for each operation, the *workload class* for each of the three resources (CPU, disk I/O and network bandwidth). A *workload class* aggregates workloads with similar demands for a particular resource. The *workload class* represents the usage intensity level for that resource. The usage intensity levels are identified by monotonically increasing numbers, up to a given superior bound. We associate to each level, a quantitative workload value. This mapping translates a *workload class* into a quantitative workload.

We also consider an enriched workload specification, that exploits the operation parameters. Such an approach claims for workload functions, to be generated for the service operations. A workload function, associated to a given operation, takes as parameter an Access object and returns a Workload object, containing three integers that represent, respectively, the CPU workload, the disk I/O workload and the network workload that the operation requires for its processing.

```
<performance_contract>
  <requirement operations="" workload="(2000, 800, 0)" bound="80 sec"/>
</performance_contract>

<performance_contract>
  <requirement operations="" workload_class="(2, 2, 0)" />
</performance_contract>

<interface>
  <operation id="O1" signature="void f(int n)">
</interface>
<performance_contract>
  <requirement operations="O1" workload="((2*n*n*n)/10000, 0, 0)"/>
</performance_contract>
```

Figure 7.2: Examples of performance contracts

Figure 7.2 shows three examples of *performance contracts*. The former performance contract is attached to a service, whose operations generate a workload equal to 2000 MFLOPS and 800 MB transferred to/from the disk. This contract contains a single requirement, which claims for a replica expected to provide a response time inferior or equal to 80 seconds.

In the second performance contract, the operations workload is specified by means of a workload class, containing the degree of the CPU usage and the degree of the disk I/O usage. This contract also contains a single requirement, which claims for a replica expected to provide the best possible response time.

The latter performance contract is attached to a service, containing an operation, parameterized by an integer. The operation workload is specified by means of an arithmetical expression, which takes into account the operation parameter. This contract contains the same requirement, as the previous one.

7.3 Genericity support

The Metric object We abstract a given metric by the object Metric, in order to treat all supported metrics uniformly within the predicates composing the replica selection criterion. These predicates are checked for a list of hosts, independently of the metrics that they encapsulate. We define a set of abstract performance levels, to be particularized for each metric. These levels are: *bad*, *reasonable*, *good*, *very good*, *excellent*. Each performance level has associated a range of values. For a particular metric, the performance levels are specified by means of five thresholds, which divide the domain of values into five successive ranges. The Metric object contains: the metric name, the domain values, the measurement unit, the comparison operator ("$<$", "$>$" or "$=$") and the thresholds defining the performance levels.

The DerivedMetric object We perform statistical computations, in order to predict the measures expected for a given metric in the near future (defined by a given time period), from the measures collected previously for that metric. Such predictions have already been proposed in Network Weather System [68]. They detect the resources that have been intensively used recently, so as to avoid using them for other requests, as the performance of those requests would decrease dramatically. We use standard statistics, such as: the mean value, the median value [24] (as in Network Weather System), the most frequent measure and the most recent measure.

A statistical computation is reified by the DerivedMetric object, containing: the history of measures to be taken into account, the computation type, the (optional) parameters needed by the computation and the comparison operator between the metric values. For example, if the computation counts for the percentage of requests that perceived unsatisfactory response time, the parameters encapsulate the threshold on the valid response time values. The measures history is defined as a time period or as the number of measures to be considered. The computation type indicates how to aggregate the measures in the history, by using standard statistics. In these cases, parameters is null.

Examples of derived metrics include: *the average value over the CPU utilization measures collected in the last hour*; *the percentage of response time values above 7 seconds observed in the last 20 minutes, among all Web accesses.*

For all the statistics, the computation of the derived metric value relies on two main methods: addMeasure() and getSample(). The former method includes a new measure within the computation of the derived metric value. The latter method returns the derived metric value, at the end of the computation.

The DerivedMetric object is specialized for each computation type (most recent measure, mean measure, most frequent measure, median measure), with the data members needed by the computation. In particular, the object specialized for computing the mean, contains as data members the number and the sum of all measures considered. At the end of the computation, the method getSample divides the sum by the number of measures considered (if this number is positive).

The object, corresponding to the most frequent measure and to the median measure, contains as data members an array with the measures considered together with the frequency of their occurrence. If the computation requires the most frequent measure, the method getSample returns the measure with the maximum frequency. If the computation requires the median measure, the method getSample ranks the measures and returns the measure, whose frequency satisfies the 50% percentage, with respect to the number of all measures, that precede it in the rank.

In the next paragraph, we define the *metric-based predicate*, that represents the building brick for defining replica selection and rebinding criterion.

Metric-based predicates We define two types of metric-based predicates: *best-effort predicates* and *relative predicates*. We further classify the *best-effort metric-based predicates* in two types: *best value* and *best level*. The former type claims for the best value, determined among a set of measures. The latter type claims for the metric values within a particular performance level. A best-effort metric-based predicate is defined as a vector with three elements: the *metric name*, the *derived metric* and the performance level. The latter parameter is absent, in the case of a predicate of type *best value*. A *relative metric-based predicate* defines the valid values, as those below a given threshold. A relative metric-based predicate is a vector with three elements: the *metric name*, the *derived metric* and the *bound*.

Abstract replication criterion A replication criterion is expressed as a boolean expression, which consists of conjunctions of *metric-based predicates*. The criterion is associated to a pair *(replica, client)* or to a pair *(replica group, client)*. In the former case, the predicates are checked for the replica and the client. In the latter case, the predicates are checked for each replica, within the group, and the client. Examples of replica groups include: all available replicas, all available replicas within a given domain and the replicas proximal to a given client, where the proximity metric counts for the number of AS hops.

In our work, we study the criteria guiding the replica selection and the rebinding decisions. An example of replica selection criterion is:
$RT\_estimator < 120ms$,
where the metric $RT\_estimator$, approximating the response time, has been defined in chapter 6. An example of rebinding criterion is:
$the\ percentage\ of\ unsatisfactory\ requests\ \geq\ 80\%$,
where a request is considered as unsatisfactory, if its perceived response time exceeds $50ms$ and the percentage is computed over all the requests submitted by the client to the current replica, during the current session.

7.4 Replica selection criterion

7.4.1 Inferring the replica selection criterion

The algorithm for inferring the replica selection criterion is depicted in Figure 7.3. We apply it for every requirement, attached to a particular operation,

```
get res_i, s.t. st_i/(st_1 + st_2 + st_3) ≥ 1/3, i ≤ 3

if the requirement is response time ≤ b
then {
  RT_estimator-based criterion = RT_estimator ≤ b
  resource capacity-based criterion = ∧_{j=1}^{i} (res_j.capacity is "good")
} else {
  RT_estimator-based criterion = best RT_estimator
  resource capacity-based criterion = best res_1.capacity, where st_1=max{st_1, st_2, st_3}
}
```

Figure 7.3: Inferring the replica selection criterion

within the performance contract. We distinguish between the case of a relative performance requirement and the case of a best-effort performance requirement. For each operation, we generate two replica selection criteria: an *estimation-based criterion* and a *relevant resources-based criterion* from the performance requirement attached to that operation. The former relies on the response time estimator, presented in chapter 6. The latter relies on the capacity of the resources, needed for processing the operation calls. We obtain a table, wherein each operation is attached a replica selection criterion.

7.4.1.1 Response time estimator-based criterion

The case of a relative requirement A relative performance requirement is formalized by: *response time* $\leq b$, where b is the bound delimiting the valid values (as specified by the service supplier in the performance contract). A request is unsatisfied, if the value of the response time, perceived for this request, exceeds the bound b. This relative requirement is translated into the selection criterion, containing one relative predicate, that imposes the bound b on *RT_estimator*. The response time estimator-based selection criterion is formalized, as follows: $RT\_estimator \leq b$.

The case of a best-effort requirement A best-effort requirement on response time is translated into the selection criterion, containing one best-effort predicate on *RT_ estimator*.

7.4.1.2 Resource capacity-based criterion

We associate to the *response time* one or several base metrics, representing the capacity of the resources that impact the response time. These base metrics are inferred from the service workload, as shown in the following.

We consider two types of workload specifications: quantitative and qualitative. Let (cpu_wk, io_wk, bw_wk) be a quantitative workload specification. We convert the individual components of the workload in the service times, that are expected from a given reference host. We obtain the services time, noted cpu_st, io_st, bw_st. We say that res_i is a highly-used resource, if:
$st_i/(st_1 + st_2 + st_3) \geq 1/3,\ i = 1,\ 3$. If the workload is specified qualitatively, by means of a *workload class* attached to each resource, we convert the three workload classes into a quantitative workload and proceed as above.

The case of a relative requirement A relative requirement on the response time is translated into *best level* predicates, imposing the performance level *good* on the capacity of each highly-used resource. In this case, the selection criterion becomes a logical expression, relating by the boolean operator "and", the *best level* conditions on resource capacities.

The case of a best-effort requirement A best-effort requirement on response time is translated into the selection criterion, containing one best-effort predicate on the capacity of the most intensively used-resource. In this respect, we compute the maximum between st_1, st_2 and st_3 and we consider the corresponding resource, as the one that is the most intensively used.

7.4.1.3 The choice of the appropriate criterion at run-time

The choice of the right criterion is performed at run-time. The choice depends on the current degree of the resource utilization, on the utilization variation and on the requests response time, compared to the estimation cost. The selection criterion must exploit a response time estimator, when the resources are shared by several users. However, when the resources are under-utilized, the resource capacity-based criterion is the appropriate one. This criterion has minimal cost, as the capacities are configured statically. This criterion should also be used, when the resource utilization varies too rapidly. In this context, it is not possible to obtain sufficiently accurate utilization measures, needed by the response time estimator. Also, the selection criterion containing a best-effort requirement on CPU capacity is appropriate in the case when the response time of the requests is close to the delay induced by the response time estimation.

7.4.2 The selection policy objects

In order to manipulate the replica selection criteria, we introduce two objects, called ReplicaFilter and Ranker. They correspond to relative, respectively, best-effort performance requirements. The two objects are presented in the next two paragraphs.

ReplicaFilter A ReplicaFilter object encapsulates a table, wherein each operation, of the replicated service, is attached its workload and a replica selection criterion. This criterion is inferred from the relative performance requirement, specified for that operation within the performance contract.

Each criterion is expressed by one of the following two data members. The former data member contains a relative predicate on the *RT_estimator* metric, associated to the response time. The predicate contains the response time threshold, that is required by the calls to that operation. The latter data member is a vector, containing a list of relative metric-based predicates, related by the boolean operator "and". These predicates are defined on the capacities of the resources on which the response time of the operation depends.

The ReplicaFilter provides the method filter, that takes as parameters: the service name, a list with the replicas IP addresses and the client identifier. The array parameter may be replaced by a map, wherein the replicas are grouped by the domains, to which their hosts belong. The result returned by the method is an array with integers. The first element in this array represents the number of satisfactory replicas. It is followed by the indexes of satisfactory replicas, matching their positions within the array parameter. The list of satisfactory replicas may be ranked, by the values of the $RT\_estimator$ metric. A variant of the filter method filters the replicas relatively to a reference replica, such as only the replicas better than this reference replica are checked against the replica selection criterion.

Ranker A Ranker object encapsulates for each operation, the replica selection criterion, inferred from the best-effort performance requirement, attached to that operation. The criterion is expressed by one of the following two data members: a best-effort predicate on $RT\_estimator$ or a best-effort predicate on the capacity of the resource, that is the most intensively-used by that operation.

The Ranker provides the method rank, that has the same list of parameters as the method filter of the Replica Filter. The method ranks the replicas by the metric values and returns the index of the best replica. A variant of the rank method takes a supplementary parameter that is a reference replica. It ranks only the replicas better than that reference replica (i.e. their metric values are better than the metric value determined for the reference replica).

7.5 The rebinding criterion

7.5.1 Inferring the rebinding criterion

The algorithm for inferring the rebinding criterion is depicted in Figure 7.4.
Definition We define a metric, called $degradated\_RT$, that counts the percentage of requests that didn't satisfy the response time requirement (i.e. expressed as a threshold imposed on the *response time*), from the beginning of the client connection to his/her replica. The value of $degradated\_RT$ is updated each time the response time of a request violates the requirement. We define the rebinding criterion by bounding the value of $degradated\_RT$.

We formalize a relative requirement as follows $response\, time \leq b$. In the case of a best-effort requirement, the threshold b is defined as equal to $1.5 * avg\_rt$, where avg_rt represents the average value, computed over the response time measures, observed by all the client requests.

Both in the case of a relative requirement and of a best-effort requirement, we formalize the rebinding criterion by:
$degradated\_RT \geq percentage$,
where $percentage = 100\% - percentage\_requests\_ok$ and $percentage\_requests\_ok$ is a parameter pre-defined in our system. It denotes the percentage of requests that satisfied the threshold b, from the beginning of the session. We set the value of $percentage\_requests\_ok$ to 80%. We also set a threshold on the minimal number of requests, that should be considered when checking the performance contract. This threshold helps dealing with transient

```
if the requirement is response time ≤ b
then
  degradated_RT = 100*(nb of requests with response time > b)/(nb of all requests)
else {
  avg_rt = compute average value over all the response time measures
  degradated_RT = 100*(nb of requests with response time > 1.5*avg_rt)
}
rebinding criterion = degradated_RT ≥ 20%
```

Figure 7.4: Inferring the rebinding criterion

overload conditions, at the beginning of the interaction between the client and the selected replica.

7.5.1.1 Predefined rebinding criteria

We consider one predefined rebinding criterion, needed in order to limit the effects of erroneous replica selections on the clients' experience. This criterion is defined as follows:
$if\ during\ the\ client\ session,\ the\ first\ a\ consecutive\ requests$
$have\ unsatisfactory\ response\ time,\ then\ rebind\ the\ client\ to\ another\ replica.$
The criterion is configured by a bound a on the number of consecutive requests, for which the performance contract has been violated.

7.5.2 The rebinding policy objects

We encapsulate the rebinding criterion inferred from the performance contract within a Checker object. The rebinding criterion is expressed as a relative predicate on the *degradated_RT* metric. The Checker provides the method check with the same list of parameters as the method filter of the ReplicaFilter object. The method checks if the predicate is verified for each pair (replica in the list, client). A variant of this method checks the predicate only for the replicas better than a reference replica, given as parameter.

7.6 The replica selection and rebinding system

7.6.1 The Replica Manager component

Each replicated service is attached a Replica Manager component. This component has three main roles. The former is to control the evolution of the replica group (by adding, migrating and deleting replicas). The second is to bind the clients to the appropriate replicas, that could satisfy the performance contract. The latter is to monitor the performance perceived by the clients from the replicas that they have been assigned, and to rebind them to another replicas, when the current ones become unsatisfactory. Our work target only the latter two functionalities.

Data members A Replica Manager instance contains two categories of data members. The data members, in the former category, characterize the service, including: its name, the suffix and the prefix that should be added around the host address so as to construct the replica reference (that is returned to the requesting client).

The data members, in the latter category, include the policy objects needed to resolve the replica selection and the rebinding decisions and the ReplicaRepository. The policy objects include: a ReplicaFilter or exclusively a Ranker object, responsible for the replica selection decision and a Checker object, responsible for the rebinding decision.

The ReplicaRepository object maintains for each service: the list of domains containing replicas, the list of replicas grouped by domains and the distances between all domains and the domains with replicas. Each available replica is described by the timestamp of its creation time instant, its IP address, the reference by which the clients interact with the replica and the timestamp indicating the last time instant when this replica has been selected for a client.

The Replica Manager implements the replica selection algorithm (described in section 7.6.2), in order to provide to each client the suitable replica. The Replica Manager performs the replica selection, before the first access of the client to the replicated service and during the interaction of the client with the replica to which he/she has been bound, when a given percentage of the client requests perceived unsatisfactory response time values. If this is the case, then the client sends a rebinding request to the Replica Manager. In order to process rebinding requests, the Replica Manager implements the rebinding algorithm, described in section 7.6.3.

7.6.2 The replica selection algorithm

The replica selection algorithm is described within the primitive select, that takes three mandatory arguments: the service name, the identifier of the operation invoked by the client and the client identifier (as shown in Figure 7.5). There is a fourth optional argument, that specifies the reference of the replica to which the client has been previously bound. The presence of this argument makes the difference between a selection and a re-selection request.

We designed a scalable greedy selection algorithm, where the suitable replica is determined only among a subset of replicas, which are proximal to the client host. We call this subset of replicas *selection scope*. The algorithm has two main stages. The former performs the coarse-grained selection, by determining the *selection scope* associated to the client. The latter performs the fine-grained selection, by applying the replica selection criterion for each replica determined in the previous stage. The following two sections detail these two stages.

7.6.2.1 Coarse-grained selection

A *selection scope* contains the subset of replicas, susceptible to satisfy the performance contract for a given client. This subset contains replicas that are proximal to the client, where the proximity distance is defined in number of AS hops.

The coarse-grained selection stage is depicted in Figure 7.5, within the primitive getSelectionScope. This primitive determines the selection scope for a *service* and a *client* requiring access to that service. Its parameter *perc* specifies the percentage of replicas to be included within the selection scope, among all available replicas of the service. The underlying algorithm works as follows. Firstly, it determines the domain CD to which the client belongs. If the number of the replicas in CD already satisfies the required percentage, the algorithm returns CD. Otherwise, the selection scope, noted SD, is initialized with CD and is augmented incrementally in successive steps. Each step determines the domains that are closest to CD and that haven't been considered yet in the selection scope. The domains are ranked, in descending order, by the number of replicas. The domains are included successively in SD, until the required percentage, on the number of replicas contained in SD with respect to the number of all replicas, is reached. The complexity of this algorithm is n^3, where n is the number of available domains.

Using *selection scopes* has two main benefits. Firstly, the traffic, generated by requests and replies, is localized within proximal ASes. Secondly, the replica selection delay is reduced, as the selection criterion is checked only for a subset of replicas. The algorithm for computing the *selection scopes* can be customized with the pertinent proximity metric. By default, this metric is the number of AS hops. However, other metrics such as the maximal bandwidth may be useful, according to the underlying selection criterion. In these cases, the distance between two domains is computed as the average over the maximal bandwidth between (all) pairs of hosts, that belong to the two domains.

7.6.2.2 Fine-grained selection

The fine-grained selection stage contains three steps. In the first step, if this is a re-selection request, the algorithm checks if the current replica satisfies the replica selection criterion. If this is the case, the algorithm stops and returns null. Otherwise, it proceeds to the second step, where it determines the list of replicas contained in the selection scope. In the third step, it applies the replica selection criterion for each replica determined in the previous step. In the case of a relative criterion, the algorithm filters the list of replicas, that satisfy the criterion with respect to the client. Among the satisfactory replicas, it chooses the one, for which the most recent binding has the smallest timestamp value. This heuristic aims to avoid that replicas remain unused, when there are incoming selection requests, for which those replicas revealed to be appropriate. In the case of a best-effort criterion, the algorithm ranks the replicas by their values, determined for the metric contained within the criterion. It returns the first-ranked replica to the client. If the client is already bound to a replica, then the algorithm takes into account only the replicas better than that replica, when performing the filtering/ranking of replicas.

```
getSelectionScope(service, client, perc) {
  let D_i, i = 1, n the list of available domains
  nb = nb_replicas(service)
  CD = domain(client)
   crt_nb = nb_replicas(service, CD) //crt_nb=the current number of replicas contained in the
selection scope SD
  if (crt_nb ≥ (nb*perc)/100)
    return CD

  SD = {CD}; k = 1
  do {
    get D, s.t. distance(CD, D)=min{distance(CD, D_i)/(distance(CD, D_i) ≥ k)∧(D_i ∉ SD), i = 1, n}
    SD = SD ∪ D
    crt_nb = crt_nb + nb_replicas(service, D)
    k = distance(CD, D)
  } while (crt_nb ≤ (nb*perc)/100)

  return SD
}

enum CriterionType { relative, best-effort }
select(service, operation, client [, prev_replica]) {
  (type, criterion) = getSelectionCriterion(service)
  if (prev_replica != NULL) {
    ok = Checker->check(service, operation, prev_replica, client, criterion)
    if (ok) return NULL
  }
  scope = getSelectionScope(service, client)
  replicas = getReplicas(service, scope)
  switch (type) {
    case relative:
     replicas = ReplicaFilter->filter(service, operation, [prev_replica,] replicas, client, criterion)
     if (replicas) {
       selected = getUnused(replicas)
       if (selected == prev_replica) selected = NULL
     }
     break
    case best-effort:
     selected = Ranker->rank(service, operation, [prev_replica,] replicas, client, criterion)
  }
  return selected
}
```

Figure 7.5: The replica selection algorithm

7.6.3 The rebinding algorithm

A rebinding request is handled by the primitive rebind (Figure 7.6), that takes five parameters: the number of requests for which the threshold on the response time has been respected, the number of unsatisfied requests, the identifier of the previously called operation, the client identifier and the reference of the current replica to which the client is bound. The main role of this primitive is to determine if the rebinding criterion is satisfied, and to select another replica, better than the current one, in the affirmative case. The primitive consists of three main steps.

The first step determines if the current binding is a false positive, by checking that no request has been satisfied and that the request number exceeds the given threshold. If the current binding isn't a false-positive, the second step checks the rebinding criterion, inferred previously from the performance contract. The third step performs the rebinding, if it is needed. This step invokes the primitive select, that looks for another replica that could potentially satisfy the selection criterion, and that is better than the current replica. If such a replica has been found, its reference is returned to the client.

```
rebind(nb_ok, nb_nok, operation, client, replica) {
  if (!nb_nok) return NULL
  need_rebind = ((nb_ok == 0) and (nb_nok ≥ a))
  if (!need_rebind)
     need_rebind = Checker->check(service, replica, client)
  if (need_rebind) {
    ref = select(service, operation, client, replica)
    return ref
  }
  return NULL
}
```

Figure 7.6: The rebinding algorithm

7.7 Experimental evaluation

In this section, we present the experimental results proving the accuracy and the scalability of our replica selection approach and the benefits of re-selecting the pertinent replica at run-time. We performed emulation-based experiments, where we run a Replica Manager instance, a Consistency Manager instance on each replica, a Host Monitor instance on each replica host and a Metrology Server in each domain. This realistic experiments helped us show the feasibility of our replication approach within a concrete Replicated Service Hosting System.

7.7.1 The replica selection accuracy

We studied experimentally the accuracy of our replica selection approach, in the case of a best-effort performance contract. We considered different frequencies with which the replica selection requests arrive at the Replica Manager. We set the threshold on *measure discrepancy* (concept introduced in section 6.8.3) to 20%.

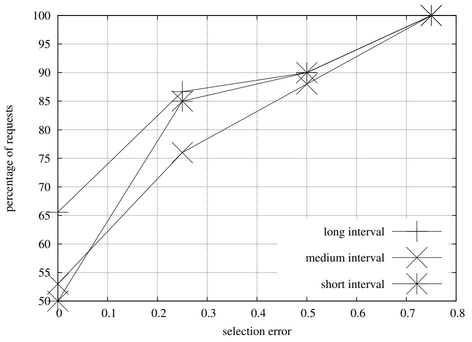

Figure 7.7: Cdf of the selection errors

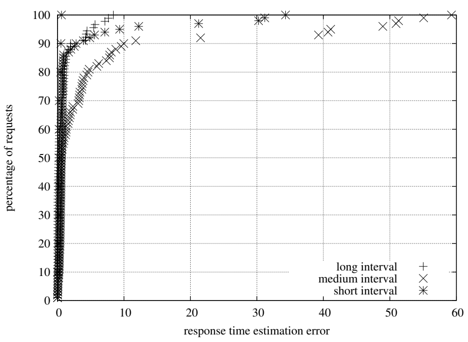

Figure 7.8: Cdf of the estimation errors

7.7.1.1 Experimental setup

We consider that on every replica, there is a Workload Simulator, which emulates a real service, as described in section 6.7.1. For each service request, the Workload Simulator generates an equivalent workload (by means of matrix multiplications, disk I/O operations, socket I/O operations). It executes this equivalent workload and logs the service times and the waiting times obtained. These logs are analyzed by the Replica Manager in order to evaluate the pertinence of its replica choices, when the requests finish. In our experimentation scenario, we consider a set of 4 replicas, which we emulate by running a Workload Simulator instance on 4 different machines. Two of them are Pentium 4 with 3GHz, 900MB RAM, running Linux and the other two are Pentium 3 with 1GHz, 256MB RAM, running Linux.

7.7.1.2 Experiments description

We simulated the arrival of successive clients at the Replica Manager and the execution of their requests by the replicas to which they have been bound. Precisely, we implemented *a virtual client* who generates series of 100 replica selection requests, with various workloads. Each replica selection request with the workload *wk* arrives at the Replica Manager, which processes it by selecting the appropriate replica *r* among the 4 available replicas. This replica is returned to the client. The virtual client invokes the service operations at the selected replica *r*, by generating requests with the workload *wk*. The Workload Simulator, running at *r*, simulates each request, by executing a workload equal to *wk*.

We maintain 3 types of logs: one log at the Replica Manager, one log at each Workload Simulator and one log at each Host Monitor. The Replica Manager writes an entry in its log, after each replica selection request, that it processes, by selecting the suitable replica. This entry contains the timestamp indicating to the selection instant, the selected replica, the response time estimated for this replica and the identifier of the session (corresponding to the current binding of the client to his/her replica).

The Workload Simulator writes an entry in its log, for each workload that it executes. This entry contains the identifier of the session to which the request belongs and the response time measured by executing the workload of the request. Simultaneously, each Host Monitor writes in its log, the utilization measures that it collects.

When the execution of the requests ends, the Replica Manager analysis the content of its log and the content of the logs maintained by the Workload Simulators and by the Host Monitors. Precisely, for each request executed by the replica r, the Replica Manager performs the following two actions. Firstly, it obtains from the Workload Simulator running at the host of r, the response time measured by executing the workload generated by the request. Secondly, from the logs maintained by the Host Monitors, it obtains the utilization values measured at the time instant, which is closest to the replica selection instant. Then it ranks in ascending order the replicas according to their response time values, estimated using those utilization measures. The Replica Manager computes for each request: the selection error and the estimation error. It computes the selection error, by dividing the position of r in this ordered sequence, by the number of replicas. It computes the estimation error of r as the difference between the measured response time and the estimated response time, divided by the measured response time. We aggregate these data for all the requests, in order to compute the cumulative distributions of the selection error values and of the estimation error values.

We represent the results in two graphs. In Figure 7.7, we plot on x-axis the selection error values and on y-axis the percentage of requests, for which the selection error was smaller than the corresponding x-axis value. In Figure 7.8, we plot on x-axis the estimation error and on y-axis the percentage of requests, for which the estimation error was smaller than the corresponding x-axis value.

We considered 3 time intervals, delimiting the requests arrivals at the Replica Manager: a *long interval* (varying from 15 to 20 seconds), a *medium interval* (varying from 5 to 10 seconds) and a *short interval* (varying from 1 to 5 seconds). These intervals were chosen according to the dissemination interval (defining the frequency of measures dissemination). Precisely, the long interval is larger than the dissemination interval, the medium interval is close to the dissemination interval, and the short interval is smaller than the dissemination interval. These experiments show two main positive results. The former result confirms that the accuracy of response time estimation and, as a consequence, the accuracy of the replica selection is satisfactory. In all the 3 experiments, 50% of requests, have the estimation error inferior to 0.6. Also, for 50% of the requests, the replica selected was the best one, which could have been determined if the real measures have been known.

The latter result shows, as expected, that the selection error is much higher when the requests frequency follows the short interval. Precisely, in this case, 50% of requests have an estimation error inferior to 0.2 and 65% of requests have been allocated the best replica. This result indicates that it is advisable to adapt the dissemination interval to the requests frequency, for example, by measuring the CPU utilization and the disk I/O utilization, after each replica selection performed for a client.

7.7.2 The replica selection scalability

We studied experimentally the scalability of our replica selection system, with respect to the number of the domains containing replicas and with respect to the number of replicas within a given domain. In this respect, we evaluated separately the two main stages of the replica selection: the coarse-grained selection, corresponding to the computation of the selection scope, and the fine-grained selection, corresponding to the evaluation of the replica selection criterion for each replica within the selection scope. We run a Replica Manager instance on a machine Pentium 4 with 3GHz, 900MB RAM.

The coarse-grained selection delay In order to evaluate the scalability of the coarse-grained selection, we varied between 1 and 100 the number of domains containing replicas. For a given number of domains, we performed 22 replica selection requests, for which we measured the delay of computing the selection scope. We computed the average delay (by ignoring the minimum and the maximum delays). We show the results in Figure 7.9. One can see that the delay increases linearly with the number of domains, and the maximum delay, perceived for 100 domains, is insignificant (below 0.5ms).

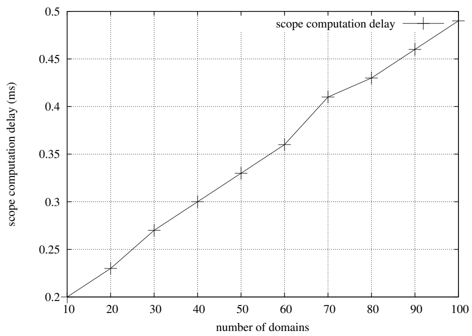

Figure 7.9: The delay of coarse-grained replica selection

The fine-grained selection delay In order to evaluate the scalability of the fine-grained selection, we considered that the selection scope contains a single domain. The rationale behind this simplified assumption is that the Replica Manager estimates the response time expected from replicas belonging to different domains in parallel. Precisely, the number of threads is equal to the number of domains containing replicas. We varied between 1 and 100, the number of replicas contained in the single domain of the selection scope. For a given number of replicas, we performed 22 replica selection requests. We measured the selection delay for each request and we computed the average delay (by ignoring the minimum and the maximum delays). We show the results in Figure 7.10. One can see that the replica selection delay increases linearly with the number of replicas. However, the maximum delay, that is obtained for 100 replicas, remains reasonable small (below 160ms).

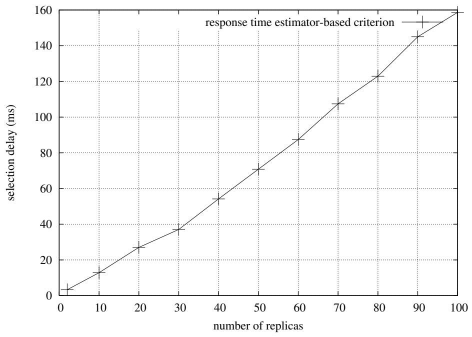

Figure 7.10: The delay of fine-grained replica selection

These experiments confirm that the selection delay is almost insignificant with respect to the average response time of the requests to the services, that we expect to replicate.

We also measured the delay of handling a rebinding request, when rebinding to a new replica is not necessary. We obtained an insignificant delay of 0.13ms.

7.7.3 The benefits of rebinding

We study experimentally the benefits of re-selecting the pertinent replica for a client at run-time, when the current replica provides unsatisfactory response time. In this respect, we run TPC-W [72] with the two profiles: browsing-mix and shopping mix. In both cases, the performance contract requires a response time threshold equal to 50ms.

We used four machines named *canardo*, *haplin*, *pataclop* and *profi*. *canardo* and *pataclop* are Pentium 3 with 1GHz, 256MB RAM. *haplin* and *profi* are Pentium 4 with 3GHz, 900MB RAM. The replica is selected initially and at run-time, among the four replicas available on the machines enumerated above. We varied the load of each replica host, by running in background a shell script, developed in our team by N. Gibelin. This script launches an arbitrary number of processes for an arbitrary period of time.

We used a client-side proxy developed in our team by I. Chabbouh [8], in order to send the requests to the Web server and to collect the measures of response time, experienced by the clients. This proxy performs the average of response time perceived by five successive requests, and it sends the computed value to the Replica Manager.

When performing these tests, we used an optimistic concurrency control, where the replicas have been accessed in isolation. We didn't use the replica consistency management available in Mysql [38], because of the absence of a mechanism for resolving the conflicts between the updates accessing concurrently the same item.

The replica selection criterion, that we used, contains a best-effort condition on the CPU utilization. The rebinding is performed at run-time when of the two situations occur:

- The first 3 requests of the session perceived response time values larger than 210ms.

- 20% of the client requests observed response times values larger than 210ms. At least 6 requests are taken into account, so as to deal with transient loads.

Each graph shows the results obtained by running all the requests within a complete TPC-W session. Precisely, we plotted the cumulative distribution of the average response time values. On x-axis we plotted the average response time values, and on y-axis we plotted the percentage of requests, with the response time value smaller than the corresponding x-axis value. On each graph, we plotted two curves, one curve representing the static binding, and one curve representing the dynamic rebinding.

Figures 7.11 and 7.12 show the results of experiments for the scenario browsing-mix. In the case of the static binding, the replica selected is *canardo* in Figure 7.11 and *pataclop* in Figure 7.12. When rebinding is used, there is an increasing of 40% of the requests which satisfied the response time threshold of 50ms. However, when the replica selected statically is *haplin* or *profi*, there is no benefit of using rebinding, as these machines are the most powerful ones.

We repeated the same experiments in the case of shopping-mix. Figures 7.13 and 7.14 show the corresponding results, when the replica selected initially is *canardo*, respectively *pataclop*. In both cases, when rebinding used, there is an increasing of 50% of the requests which satisfied the response time threshold of 50ms.

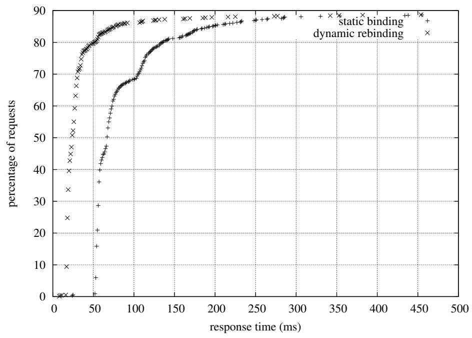

Figure 7.11: Browsing-mix, where the replica selected statically is *canardo*

7.8 Conclusion

We developed a protocol, aimed to select the suitable replica for each client, at the beginning of his session and dynamically. This protocol makes use of the response time estimator presented in the previous chapter. We obtained experimentally three main results. The former result shows the accuracy of the replica selection, if the accuracy of resource utilization measures respect a given staleness threshold (satisfied by selecting the appropriate monitoring interval and dissemination criteria). The second result confirms the scalability of our

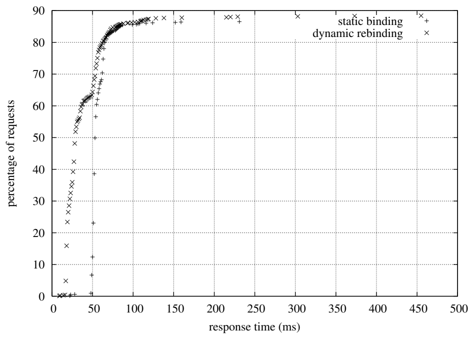

Figure 7.12: Browsing-mix, where the replica selected statically is *pataclop*

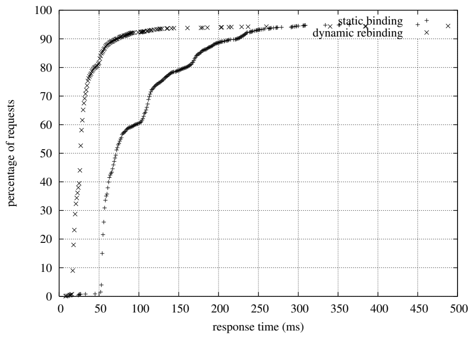

Figure 7.13: Shopping-mix, where the replica selected statically is *canardo*

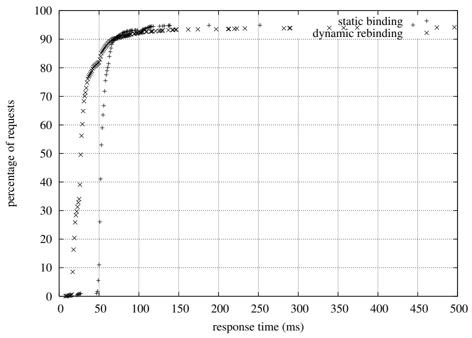

Figure 7.14: Shopping-mix, where the replica selected statically is *pataclop*

replica selection approach, by showing that increasing the number of replicas and the number of domains has a minimal impact on the selection delay (compared to the average response time expected for the replicated services). The latter result shows the benefit of re-selecting the appropriate replica dynamically when the current replica becomes unsatisfactory. However, we obtained this gain only when the newly selected replica has a better processing performance compared to the previous one.

Chapter 8

The Overall Replicated Service Hosting System

8.1 Introduction

In order to prove the feasibility of our approach, we need to provide the prototype for the Replicated Service Hosting System, which integrates the independent building bricks implementing both the consistency guarantees and the performance improvement. The prototype has been implemented and tested on Linux 2.4 and 2.6 mandrake. This chapter assembles the components providing the service-specific consistency contract and the components providing the response time requirements within a single Replicated Service Hosting System. These components are laid out on two layers: the system layer and the service layer. The Replicated Service Hosting System is made accessible to the service suppliers and to the clients, by means of a well-defined interface, that registers a new service, deploys a new replica of a service and binds the clients to the replicas.

The rest of this chapter is structured in six sections. Section 8.2 introduces a new component called Information Repository, that maintains various data related to the services, to the replica hosts and to the components locations. Section 8.3 shows the distribution and the interaction between the system-specific components, including: Information Repository, Response Time Estimator, Host Monitor and Metrology Server. Section 8.4 presents the representation of replicated service, by means of a Server-Side Replica Wrapper and a Client-Side Replica Wrapper, automatically generated from the contract specification. Section 8.5 shows the distribution and the interaction between the service-specific components, including: Consistency Manager, Replica Manager, Server-Side Replica Wrapper and Client-Side Replica Wrapper. Section 8.6 shows how the Replicated Service Hosting System is made accessible to the service suppliers and to the clients. Finally, section 8.7 concludes the chapter.

8.2 The Information Repository component

We introduce a new system-specific component, called Information Repository. This component stores and provides access to data that characterize the replicated services (including their attached contracts) and the locations of the component instances.

Each data is associated a key, that identifies uniquely that data in the repository. The key is a string, obtained by concatenating three strings of fixed length. They represent, respectively, the name of a service, a domain identifier and an IP address. For simplicity of the Information Repository implementation, we consider as domains, the Autonomous Systems existing in the Internet. The Information Repository is constructed as a Distributed Hash Table, containing the mappings of keys to data. We implemented it, using the Chord P2P system [41]. In this respect, we consider the P2P overlay, containing the potential replica hosts.

The interface of the Information Repository contains two main methods: publish and lookup, that stores data into the repository, respectively, retrieves data from the repository. The method publish takes as parameters the string containing the service name, the integer containing the domain identifier, the string containing the IP address and the string containing the data. The method lookup takes as parameters the service name, the domain identifier and the IP address. The two methods rely respectively on the primitives store() and fetch(), available in the Chord distribution.
A benefit of using Chord is that it resolves the issues, such as the selection of the host, where the data will be stored and the rooting of the queries to the host that has the data assigned to the given key. Consequently, the utilization of the Information Repository becomes very easy: it claims only for the specification of the three strings composing the key and of the string containing the data, as arguments of the invocations to the publish and to the lookup methods.

8.2.1 Maintaining the locations of the component instances

In this subsection, we show how the Information Repository is used so as maintain the locations of the instances of the following components: Replica Manager, Metrology Server and Consistency Manager. The location, of each instance of the three components, contains the IP address of the host, where the instance is running, and the number of the port, on which the instance is waiting for clients' connections. We assign a key to the location of a given component instance. The key is a string, defined by concatenating three strings, as follows.

In the case of a Replica Manager instance, the former substring identifies the service that is replicated, the second substring contains the identifier of the domain, to which this instance is attached and the latter substring has the predefined value *"0.0.0.2"*. The Information Repository provides the method lookupRMRef, which takes as parameters the service name and the domain identifier. The method returns the reference of the Replica Manager, instantiated for that service in that domain.

In the case of a Metrology Server instance, the former substring is "Metrology-

Server", the second substring contains the identifier of the domain to which this instance is attached and the latter substring has the predefined value *"0.0.0.1"*. The Information Repository provides the method lookupMSRef, which takes as parameter the domain identifier. The method returns the reference of the Metrology Server, instantiated in that domain.

In the case of a Consistency Manager instance, the former substring identifies the replicated service, the second substring has the predefined value *"-1"* and the latter substring contains the IP address of the replica host, where this instance has been created. The Information Repository provides the method lookupCMRef, which takes as parameters the service name and the replica IP address. The method returns the reference of the Consistency Manager, instantiated for that service, on that replica host.

The mappings of keys to locations are distributed across the replica hosts, that compose the P2P overlay. The usage of high-level identifiers determines dynamically, the port to be used by a particular component instance, according to the network configuration available on its replica host. In particular, different instances of the Consistency Manager component may use different ports, as available on their replicas hosts.

8.2.2 Maintaining the service-related descriptions

As we have already mentioned, in order to be replicated, each service needs the specification of the consistency contract and of the performance contract. The two specifications are given by the service supplier within an XML file. The service supplier also provides the service deployment description, in a separate file. This description contains the informations needed by the Service Deployment component, such as the location of the archive containing the sources and the configuration files of the service [23]. We concatenate the locations of the two files in a string, that is assigned the service name, as the key. The corresponding mapping is stored within the Information Repository.

We enrich the Information Repository with two methods, called lookupDeployment and lookupContracts. Both methods takes as argument the service name, and obtain the content of the deployment description, respectively, the content of the consistency contract and of the performance contract, associated to that service.

8.3 The system-specific components

The system-specific components include: the Host Monitor, the Metrology Server (presented in section 6.8), the Response Time Estimator (presented in section 6.9), the Service Deployment and the Information Repository (as shown in Figure 8.1). There is a Host Monitor instance running on each machine that is hosting a replica or that could host a replica of any service. The Host Monitor captures the measures for dynamic metrics, including: the CPU utilization, the disk I/O utilization and the number of active processes. In each domain, there is a Metrology Server. The Metrology Server stores and provides access to the

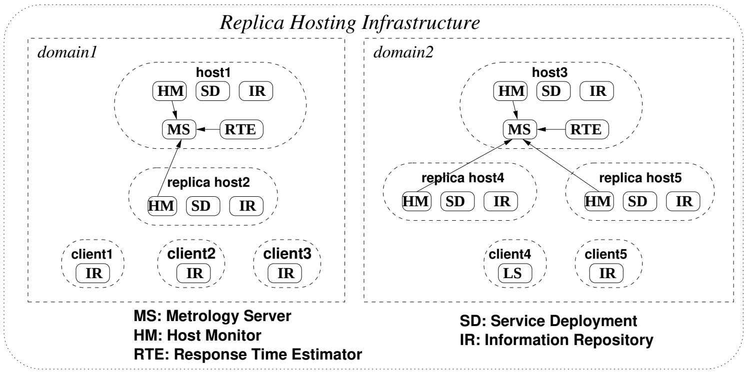

Figure 8.1: The system-specific components

measures collected on the hosts, belonging to that domain, by the Host Monitors. The Metrology Server contains a Response Time Estimator instance, that approximates the response time expected to be obtained from a given host in the domain, by executing a given workload on that host. Also, each replica host contains a Service Deployment instance (realized by N. Gibelin [23]), that deploys and instantiates on that host, a replica of any service. The Information Repository is instantiated on every replica host and on every client host.

8.4 The representation of a replicated service

Each replicated service is represented thanks to three main classes: the service implementation class, the Server-Side Replica Wrapper and the Client-Side Replica Wrapper. The service implementation class is provided by the service supplier, within the archive, containing all the sources needed, in order to instantiate the service. The Server-Side Replica Wrapper and the Client-Side Replica Wrapper provide the replicated version of any service, by enriching the service logic with the consistency management, required so as to apply correctly each operation call at all the replicas and to enforce the replicas convergence. The feature of replica consistency management is provided transparently, both for the service suppliers and for the clients. Both Server-Side Replica Wrapper and Client-Side Replica Wrapper classes are generated statically, from the XML specification of the service-specific consistency contract and of the performance contract. The following two subsections present the two classes.

8.4.1 The Server-Side Replica Wrapper

The server-side replica wrapper encapsulates the local instance of the service. It provides for each operation of the replicated service, the method <O$_i$>_apply. O_i represents a numerical identifier generated for the current operation, according to the order of the operations declaration within the XML specification of the consistency contract. The method <O$_i$>_apply takes as parameter an Access object, representing an invocation to that operation and performs as follows. Firstly, the wrapper desencapsulates the invocation's arguments from the

Access object. Then, the wrapper invokes the operation on the local service instance. Finally, the wrapper reifies the result of the operation call and inserts it into the Access object.

The appropriate <O_i>_apply method is determined from a dispatcher method apply, by extracting the identifier $<O_i>$ of the called operation from the Access object, given as argument.

8.4.2 The Client-Side Replica Wrapper

The base class The client-side replica wrapper class, associated to a given replicated service, is derived from a base class, implemented independently of the requirements of a particular service. This class contains a map, associating to each operation, its workload and the threshold on response time required to be observed by the invocations to that operation. The constructor of the base class obtains the reference of the Replica Manager instance, attached to the domain to which the client's host belongs. In this respect, the wrapper queries the local Information Repository instance, passing to it as parameters the name of the service and the number of the AS to which the client host belongs. The Information Repository returns to the client the reference of the Replica Manager, responsible for the client's AS.

The wrapper contacts the Replica Manager instance, in order to find the suitable replica for the client, before the first call issued by the client and, dynamically, when the workload class of the operation calls changes and when the threshold on the response time has been violated, for a given percentage of requests.

The base class provides the method apply, parameterized by an Access object. This method is called for each operation invocation, issued by the client and contains three actions. The first action is performed if the workload class of the current operation call changes from the workload of the previous one. If this is the case, then the wrapper contacts the Replica Manager in order to reevaluate the suitability of the current replica. The second action consists in the distant call to the replicatedAccess method of the Consistency Manager instance, associated to the replica selected for the client. The third action consists in collecting the measure of the response time observed for the operation call. The response time is approximated by the time interval elapsed between the beginning and the end of the replicatedAccess call. The perceived response time is compared against the required threshold, and the number of satisfied or unsatisfied requests is incremented, according to the comparison result. If the percentage of unsatisfied operation calls reaches a given threshold, then the client-side replica wrapper sends to the Replica Manager, a rebinding request containing the number of satisfied requests and the number of unsatisfied requests. If, by processing the rebinding request, the Replica Manager reselects another replica, then the client-side replica wrapper will redirect the client to that replica.

The Client-Side Replica Wrapper generated for a given service The client-side replica wrapper, associated to a given service, has the same public interface as the underlying service. For each operation of the service, the replica wrapper provides its replicated version, which calls the replicatedAccess method of the Consistency Manager instance, running on the replica, to which the client has been bound.

The replicated operation version has the same signature as the corresponding service operation and contains three main actions. The first action reifies the current invocation into an Access object. The second action consists in the call to the apply method of the base class, with the reified access, passed as parameter. The latter action desencapsulates the result of the invocation from the Access object, and returns it to the client.

The client-side replica wrapper is useful especially when there is some state to be maintained at the client-side. Examples of such state include the sessions, as defined in Bayou, or the response time observed by the operation invocations. The client-side replica wrapper also defines the access attributes (e.g. the caller attribute), that are known only by the access caller.

8.4.3 The generation tools

8.4.3.1 The generation tool for the Replica Wrappers

We consider two tools, called Sever-Side Generator and Client-Side Generator, used in order to generate the replicated version of each service. Both generators takes as input the XML file containing the service-specific consistency contract and the performance contract.

The Server-Side Generator generates as output two classes: the ParamFactory class (which reifies the invocations to each operation) and the Server-Side Replica Wrapper class (which applies the reified operation invocations, on the local service instance). In order to generate the two classes, the Server-Side Generator exploits the specification of the operations signatures. Precisely, in order to abstract the invocations to a given operation, the generator needs the types and the names of the parameters of that operation. The generator reifies each parameter, by a ParamWrapper object, wherein it encapsulates the parameter value. The reified parameters are inserted in the Access object, that represents the current invocation.

Also, in order to apply a refied invocation to a given operation, the generator needs the operation name, the types of the parameters and of the type of the data returned. With these informations, the generator desencapsulates the arguments from the Access object, invokes the operation, reifies the result and inserts it into the Access object.

This generator also determines the replica selection criterion and the rebinding criterion, from the specification of the workloads and of the response time bounds, assigned per operation. In this respect, it uses the algorithms that we described in sections 7.4.1 and 7.5.1.

The Client-Side Generator also generates as output two classes: the ParamFactory class (generated as in the case of the previous generator) and the Client-Side Replica Wrapper (that provides transparent access to the remote replica instance). The generator includes in the body of the Client-Side Replica Wrapper's constructor, a map that associates to each operation, its workload and the required response time threshold. The methods of the generated wrapper correspond to the service operations. In order to generate a method, associated to a given operation, the generator exploits the parameters names and the type of the returned data. The generator uses these informations, in order to reify the current invocation into an Access object (by using the ParamFactory class) and to desencapsulate the result of the distant operation call (in order to return it to the client).

```
Contract* contract = new Contract();

The tag operation is processed as follows:
  name_fct="weight_getValue(id)"
  generate the function name_fct of type WeightFct using getValue(weight)
  contract->weights[getValue(id)]=&name_fct
  if (getValue(type)=="update")
    contract->updates_ids.push_back(getValue(id))

The tag quality_of_observable_state is processed as follows:
  if (getValue(operations) {
    contract->state_quality=new PropagationDelayMonitor()
    for each op ∈ getValue(operations)
      contract->state_quality->propag_delay[op]=getValue(propagation_delay)
  } else {
    if (umetric=getValue(user-defined_metric))
     contract->state_quality= new umetric();
    else
        contract->state_quality = new TactConditionMonitor(getValue(numerical_error), getValue(order_error), getValue(staleness))
  }

The tag concurrency_control is processed as follows:
  if (getValue(type) == "pessimistic")
    contract->conc_mode = pessimistic;
  else contract->conc_mode = optimistic;
  contract->state_quality->setStabilizationDelay(getValue(stabilization_delay))

The tag concurrency_control is processed as follows:
  for each pair ∈ getValue(pair_operations) {
    name_fct="resolve_" + pair.op1 + "_" + pair.op2
      generate the function name_fct of type ResolutionFct, using getValue(condition) and getValue(resolution)
    contract->relations[pair]=&name_fct
  }

The tag dependency_control is processed as follows:
  if (getValue(dependency_type))
    contract->pred_dependencies=getValue(dependency_type)

  if (getValue(user_dependency))
    for each pair ∈ getValue(pair_operations) {
      name_fct="dependency_" + pair.op1 + "_" + pair.op2
      generate the function name_fct of type DependencyFct, using getValue(user_dependency)
      contract->user_dependencies[pair]=&name_fct
    }
```

Figure 8.2: Parsing an XML service-specific consistency contract

8.4.3.2 The generation tool for the Contract object

Figure 8.2 shows how an object Contract is generated from the XML specification of the service-specific consistency contract by the ContractFactory. We show briefly the code for processing each tag, met during the parsing of the XML specification. The value of an attribute is obtained using the primitive getValue, with the attribute name, given as argument.

Parsing the tag *operation* generates the weight function (of type WeightFct), from the arithmetical expression, specified within the attribute *weight*. The arithmetical expression exploits the numerical arguments and/or the numerical attributes of the access, in order to provide the integer weight. The association of the operation identifier to the function name is inserted in the Contract. Furthermore, if the operation is of type update, then its identifier is included in the Contract.

Parsing the *quality_of_observable_state* tag instantiates the StateMonitor object. This object could be the PropagationDelayMonitor with the propagation delay attached per group of operations, the TactConditionMonitor instantiated with the values of the metrics bounds, obtained from the tag's attributes, or a user-defined StateMonitor. Parsing the *concurrency_control* tag conducts to the generation of a resolution function (of type ResolutionFct) for each pair of operations potentially conflicting or non-commutative. The association of the pair of operations to the function name is inserted in the Contract.

Parsing the *concurrency_control* tag determines the concurrency control mode and the stabilization delay.

Parsing the *dependency_control* tag inserts in the Contract, the required predefined dependencies or user-defined dependencies. A user-defined dependency, attached to a group of operations, is represented by a boolean function, automatically generated from the boolean expression contained in the *user_dependency* attribute.

```
int weight_O1(Access& a) {
  return (a.getParam(1)->toInt())/2;
}

PairAccesses* resolve_addCourse_addCourse(Access& a1, Access& a2) {
  if (*a1.getParam(1) == *a2.getParam(1))
    if (a1.getAttrib(weight) >= a2.getAttrib(weight))
      return new PairAccesses(new Access(a1));
    else
      return new PairAccesses(new Access(a2));
  return NULL;
}
```

Figure 8.3: Examples of a weight function and a conflicts resolution function

Figure 8.3 shows an example of the weight function weight_O1, that computes the integer weight, by an arithmetical expression, wherein the value of first argument of the access is divided by 2. This figure also shows the conflicts resolution function resolve_addCourse_addCourse, generated from the XML specification in Figure 3.3, for the pairs of invocations to the operation addCourse of the e-learning service. Firstly, the function checks if the two accesses have equal values for their first argument. If true, then the function accepts the access with the biggest weight value.

8.4.3.3 The generation of the program assembling the building blocks

An important issue is how to realize the interaction between the replica consistency management code, common to all services, and the service-specific objects, such as the Contract and the Server-Side Replica Wrapper. In order to solve this issue, we instantiate the system components from a program, automatically generated for each service, by the Contract Factory component.

The function main of the program creates successively the following objects: the Contract (generated as described in section 8.4.3.2), the Consistency Manager object, the service instance and the Server-Side Replica Wrapper. The service instance is created by using the class name specified in the attribute *name*, of the tag *service*. The Consistency Manager instance is activated as a server, configured with the Contract and the three resolvers: Propagator, Scheduler and DependencyResolver.

8.5 The service-specific components

The service-specific system components include the Consistency Manager (presented in chapter 4), the Server-Side Replica Wrapper, the Client-Side Replica Wrapper (presented in section 8.4) and the Replica Manager (presented in section 7.6). The Server-Side Replica Wrapper and the Client-Side Replica Wrapper components are specialized for each replicated service, into specific classes. The underlying Consistency Manager and Replica Manager classes are general to all services, only the parameters with which they are instantiated differ from a service to another.

Figure 8.4 shows the distribution of the service-specific components. In order to represent the replica wrappers, we used the notations proposed in [61]. On each machine that is hosting a replica of a given service, there is a Consistency Manager instance and a Server-Side Replica Wrapper instance. The Consistency Manager enforces the constraints contained within the service-specific consistency contract. The Replica Manager provides access for the clients to the service, by selecting the suitable replica, at the client-connection time and dynamically, when the current replica provides unsatisfactory performance. In order to achieve scalable replica management, we instantiate, for each replicated service, one Replica Manager instance per domain.

Figure 8.4 also shows the interactions between the service-specific components, that are needed so as to reply to the client's requests and to enforce replicas convergence. The Client-Side Replica Wrapper obtains the suitable replica from the Replica Manager, responsible for the client's domain. The Consistency Manager, running on the replica, selected for the client, intercepts the client requests sent by a Client-Side Replica Wrapper, and apply them on the colocalized Server-Side Replica Wrapper. The Server-Side Replica Wrapper invokes the

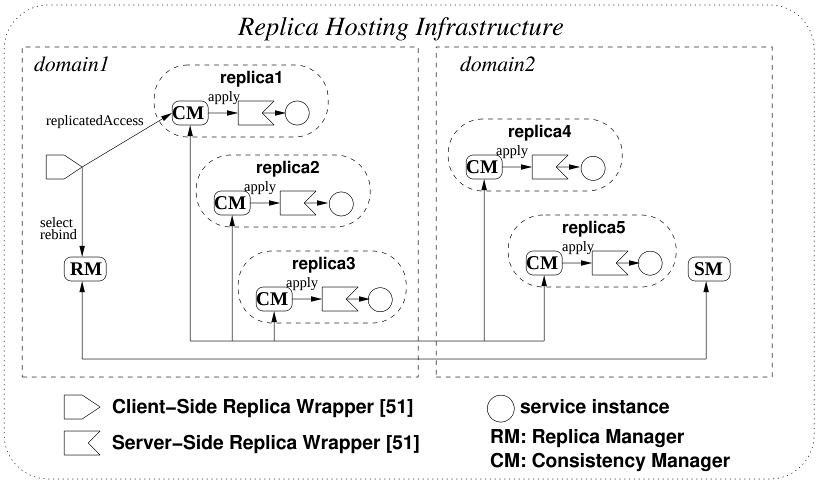

Figure 8.4: The service-specific components

requests on the local service instance. Each Consistency Manager instance cooperates with its peers Consistency Manager, in order to guarantee that the client requests are applied correctly at all the replicas, according to the service-specific consistency contract. The Client-Side Replica Wrapper also sends to the Replica Manager rebinding requests, containing the number of requests that satisfied the performance contract, and the number of requests that violated it. The Replica Manager instances also collaborate between them, in order to enforce the convergence of the replicated Replica Repository.

For simplicity of the components representation, we omitted from 8.4, the interactions between the service-specific components and the system-specific components. In fact, the Replica Manager is the only service-specific component, that uses the functionalities of the system-specific components. Precisely, it uses the Metrology Server, in order to obtain the values of the response time estimated from the available replicas and the capacities of the highly used resources.

8.6 The utilization of the Replicated Service Hosting System

The Replicated Service Hosting System is made accessible to the users by means of two modules: the service-supplier module and the clients module. Their C++ interface is showed in Figure 8.5.

8.6.1 The service supplier module

The service-supplier module contains primitives, that register a service within the Replicated Service Hosting System and start/stop the service on/from a given set of hosts. When the service-supplier wishes to register his service within the Replicated Service Hosting System, he provides the deployment description, and the description of the consistency contract and of the performance contract. The service registration contains three main actions. The first action writes the

```
//the service-supplier module
short registerService(char* service, char* deployment_descr, char* contract_descr);
short startService(char* service, short nb_hosts, char** hosts);
short stopService(char* service, short nb_hosts, char** hosts);

//the client module
short bind(char* service);
```

Figure 8.5: The interface of the Replicated Service Hosting System

```
short bind(char* service) {
  lookup consistency contract
  generate the class representing the client-side replica wrapper
  compile the class
}
```

Figure 8.6: The algorithm of bind

two descriptions in two separate files. The string concatenating the locations of the two files is attached to the service name, as the key. This mapping is stored within the Information Repository. The second actions consists in the generation of the Server-Side Replica Wrapper, attached to the service, the scheduling functions, the weights functions and the program instantiating the Consistency Manager and the Server-Side Replica Wrapper. The third action consists in the generation of the executable needed for instantiating the Consistency Manager and the Server-Side Replica Wrapper.

The primitive for starting a service, on a given set of hosts, relies on the Service Deployment's primitives, that deploy and activate a service on a given host [23]. These primitives make use of the service deployment description, which is obtained from the Information Repository, by calling its method lookupDeployment.

8.6.2 The client module

The client module contains the primitive bind, that provides the Client-Side Replica Wrapper in three main steps, depicted in Figure 8.6. Firstly, the primitive obtains the XML description of the consistency contract and of the performance contract, by calling the method lookupContracts of the Information Repository. This description is stored locally within a temporary file. Secondly, the primitive generates the class representing the Client-Side Replica Wrapper, by calling the Client-Side Replica Wrappers Generator, with the name of that file, as argument. Finally, the primitive compile the class in order to obtain the object file with which the client application will be linked.

Each client is identified by a string, that concatenates two strings, containing, respectively, the IP address of the client host and the user name with which the client is logged. This identifier is included within the operation calls, that the client initiates.

8.7 Conclusion

In this chapter, we showed the global Replicated Service Hosting System, integrating all the components needed for satisfying the consistency constraints and the response time requirements. An important contribution of our work, that we show in this chapter, is providing the consistency and the performance guarantees transparently for the clients and semi-transparently for the service providers (which need to specify only the consistency and the response time requirements). We provide this feature by integrating the service logic with the replica consistency management, transparently for the service suppliers and for the clients by using client-side and server-side replica wrappers. Each service has associated two replica wrappers: a client-side replica wrapper and a server-side replica wrapper. Both of them automatically are generated by our system from the service supplier's specification.

Part IV

Conclusion and Perspectives

Chapter 9

Summary of our Replication Approach

We proposed a Replicated Service Hosting System, customizable by service-specific guarantees, with respect to the response time expected from the replica assigned to the client, and with respect to the replica consistency. A service encapsulates data, made accessible to the clients by a well-defined interface, containing a set of operations.

9.1 Specifying and enforcing the service-specific consistency contract

The consistency contract is needed so as to preserve the service correctness and to bound the discrepancy between the state of a given replica and the ideal replica state. The service-specific guarantees are specified statically by the service supplier. The consistency contract configures a generic consistency protocol, that treats the operation calls uniformly, independently of the underlying service semantics. The constraints, contained within the consistency contract, are enforced for each operation call, transparently for the final clients. We proved that this protocol works correctly and it has a reasonable overhead.

9.2 Specifying and enforcing the performance contract

The performance contract containing requirements on response time, attached to each operation, is also specified statically by the service supplier. The enforcement of the response time requirements relies on the replica selection criterion and on the rebinding criterion, that are inferred automatically from the performance contract.

The replica selection criterion makes use of a response time estimator, or of the processing capacities of the resources needed by the service. The response time estimator is defined by means of a function, wherein each resource has associated a weight, matching the degree with which that resource impacts the response time, perceived for each request. In order to obtain this function, we used the regression-based statistical method. The replica selection criterion customizes a generic replica selection protocol, which is invoked when a client wishes to access the service.

The rebinding criterion specifies the percentage of operation calls for which the non-satisfaction of the performance contract is tolerated. This criterion customizes a generic rebinding protocol, invoked when clients send rebinding requests, notifying unsatisfactory performance.

We implemented the replica selection and the rebinding protocols, and we validated them experimentally. We obtained four main positive results. Firstly, our approach for response estimation is reasonably accurate, the values of the estimated response time are close to the real response time measure in the case of workloads, containing one or two bottleneck resources. Also, the variation of the estimated response time matches the variation of the measured response time. Secondly, our estimation approach works well even if it relies on measures with a bounded degree of inaccuracy, with respect to the actual measures. Precisely, we were able to set experimentally a threshold on the measures inaccuracy, so that the response time estimation (and consequently the replica selection) remains satisfactory. Thirdly, we showed the benefit of using dynamic rebinding over a static binding of clients to replicas. Precisely, when using rebinding, the percentage of requests that satisfied the performance contract increases significantly. Finally, we showed that the overhead of the replica selection protocol increases linearly with the number of replicas considered within the selection scope. However, the replica selection overhead, obtained when there are 100 available replicas, remains insignificant with the response time expected for the service requests.

9.3 Summary of the Replicated Service Hosting System features

The features, that our Replicated Service Hosting System provides, include:

- the flexibility of the performance contract, which matches the service workload and the flexibility of the consistency contract, which matches the service semantics;
- the customizability of the replica selection criterion and of the rebinding criterion, so as to satisfy the performance contract;
- the scalability, with respect to the domains count, replicas count and clients distribution;
- the replication transparency, so that the clients access the service using the same interface;
- the accommodation of the replicas hosts heterogeneity;
- the extensibility feature, so as to add new estimators for the response time or for other service-specific metrics (e.g. data throughput);

- semi-decentralized management of replicas, at domain level; the Replica Manager instances share only the list of available replicas;
- location-transparency of system components;
- our replication approach works for several types of services, including Data Objects and Web applications;

9.4 Discussion

Other service-specific metrics Our approach can be easily extended to support other service-specific metrics, besides the *response time*. Examples of other service-specific metrics include: the *data throughput* and the *availability*. The service supplier specifies its requirement, within the contract similarly to the case of the response time. Also, the system should provide at least one estimator for each service-specific metric. The throughput can be estimated based on the *maximal_bandwidth* and the workload of active requests (i.e. by allocating the maximal bandwidth equally among the requests that need this resource). The availability can be estimated as the minimum between the *host availability* and the *network availability*.

Open questions The replication approach that we proposed, raises several open questions, including:

1. could a response-time estimator be used as a base metric for other service-metrics, like the data throughput?
2. could the threshold on the divergence metrics be inferred automatically from the semantics of the service data?
3. when using the optimistic replication, how to provide to users, some probabilistic guarantees about the success of their tentative updates?
4. how to formalize the tradeoff between the replication accuracy (i.e. the degree of satisfaction of the performance requirement) and its cost (i.e. the measure of the resources consumed so as to satisfy the contract)?

Chapter 10

Perspectives

10.1 Consistency management under replica disconnections

10.1.1 The organization of replicas

Until now, we considered that the replicas are fully connected. Unfortunately, this is not always the case. A subject to consider as future work is the case when replicas have intermittent connections to the group, due to voluntary or involuntary disconnections. We sketch the solution we envisage, under this assumption.

We divide the replicas into two categories, called *primary-level replicas* and *secondary-level replicas*, as shown in Figure 10.1.

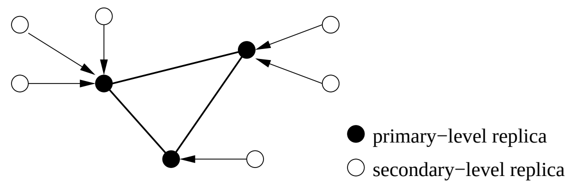

Figure 10.1: Replicas organization

The replicas in the former category are strongly connected with each other, while the others are only weakly connected. A secondary-level replica is attached to a single primary-level replica. This assignment can change dynamically. A primary-level replica has references of all other primary-level replicas and the references of the secondary-level replicas that are attached to it. A secondary-level replica has one main reference to the primary-level replica to which it is attached. It also maintains the references of the other primary-level replicas, in case when it will be attached another primary-level replica. To deal with disconnections, the consistency protocol needs several extensions, that will be presented in the next section.

10.1.2 Extensions of the consistency protocol

We distinguish between the consistency protocol performed by a primary-level replica, noted *pr*, from that of a secondary-level replica, noted *sr*.

The protocol performed by a primary-level replica *pr* maintains a log containing the scheduled updates issued by itself or by the group of secondary-level replicas attached to it. For reasons of query efficiency, we choose to divide the log into several logs, each one maintaining the accesses to a given operation, issued at a given replica.

The primary-level replicas represent the set of schedulers that perform the pessimistic concurrency control. Also, only replicas from this group compute schedules, by reconciliating updates, in the case of the optimistic concurrency control mode.

pr sends its locally-issued updates, as well as the scheduled updates, that it reconciliated (as a coordinator), to the other primary-level replicas and to its secondary-level replicas.

pr handles the updates received from a peer (in particular from the coordinator in the case of updates scheduled optimistically), as described in Section 4.4. In addition, *pr* performs a supplementary step which depends on the peer type. If the peer is a primary-level replica, then *pr* forwards the newly-received updates to the secondary-level replicas attached to it. If the peer is a secondary-level replica, then *pr* forwards the updates to the other secondary-level replicas, attached to it, and to the primary-level replicas.

pr answers a request for updates, by forwarding the request to all primary-level replicas (i.e. including itself). Each primary-level replica replies with its accepted accesses, that are absent from the received history. Then, *pr* aggregates the partial replies into a group of accesses, which it returns to the requester.

When *pr* receives a scheduling request from a secondary-level replica, it treats the update, contained within the request, as it has been issued locally. Precisely, it determines the acceptance execution order of that update, by combining the local scheduling decisions of all primary-level replicas. This scheduling decision is transmitted back to the issuer of the request.

The protocol performed by a secondary-level replica *sr* sends its updates (tentative or commutative) only to the primary-level replica, to which it is attached. The Propagator of *sr* sets a rejection flag, if the bound on the numerical error is exceeded, but *sr* can't send its updates because it is disconnected.

In the case of the pessimistic concurrency control, the group of schedulers, known by *sr*, contains only its primary-level replica. After a reconnection, *sr* resends to its primary-level replica the updates, whose propagation delay elapsed, but that are still in the outgoing state. At this time, it also gets the missed updates from the primary-level replicas.

10.1.3 Deciding primary vs. secondary-level replicas

The first replica created is a primary-level replica. The group of primary-level replicas changes dynamically, after the creation of a new replica or after the failure or the disconnection of a primary-level replica.

Each replica is characterized by two parameters, called *host availability* and *network availability*. These two parameters indicate respectively the probability with which the replica host is permanently available and the probability with which the replica host is permanently connected to the network. If the values of these two parameters are superior to predefined thresholds, and the host can communicate with all the primary-level replicas, then the new replica is a primary-level replica. Otherwise, it is a secondary-level replica and it is attached the closest primary-level replica, in terms of ASes hops number.

When a primary-level replica isn't able to communicate with another primary-level replica, it runs an algorithm for selecting the group of primary-level replicas. This algorithm works by maximizing the number of primary-level replicas, that are fully connected. In the worst case, the replicas organization is reduced to the classical primary-backup replication model [47].

When a secondary-level replica *sr* finds out that its primary-level became unavailable, it runs an algorithm that assigns to it another primary-level replica. If no primary-level replica can be contacted, this indicates that *sr* is disconnected from the replicas group.

10.1.4 The protocol correctness under disconnections

In the case of disconnections, the safety properties are provided in the same way, but the maximum delays, to reach various access states, change.

In particular, obtaining the predecessors at the initiator (i.e. the transition getExecPreds) may timeout, leading to the state *rejected*. Consequently, the delay to reach the state *ready* becomes:
$min\{timeout,\ |a.preds|\ *\ max\_propagation\_delay\}$,
where *timeout* is a system-defined parameter, stating how long the replica may wait for the predecessors of an access.

The maximum delay needed to reach the state *received* is also bounded. We have to guarantee that an update, sent by the replica r_i to the primary-level replica r_j, will arrive sooner or later at all replicas. We distinguish between the cases when r_i is a primary-level replica or a secondary-level replica, and between the cases when r_i and r_j has the same view or different views of the primary-level replicas group. We note the view of r_i by V_i and the view of r_j by V_j.

1) Let r_i be a primary replica, that calls send(a)
1.1) if $V_i == V_j$, then r_j forwards a to its secondary-level replicas. The maximum delay for a, to be received by all the secondary-level replicas, is:
$C(a).propagation\_delay\ +\ resend\_frequency\ +\ down\_interval$,

```
getPlacementScope(service [, domains_out]) {
  determine the bindings distribution (D_i, nb_bindings_i), nb_bindings_i > 0, i = 1, n.
  for i = 1 to n {
   access_frequency_i = nb_bindings_i / (1+ nb_replicas(service, D_i))
   determine the percentage bad_perf_i of unsatisfied requests in D_i, since the last replica creation
   load_domain_i = access_frequency_i * bad_perc_i
  }
  get D_k, such that load_domain_k = max{load_domain_i/D_i ∉ domains_out}, i=1, n

  return D_k
}
```

Figure 10.2: The algorithm for creating the placement scope

where *resend_frequency* defines the frequency with which a primary-level replica resends the unacknowledged updates to its secondary-level replicas and *down_interval* defines the maximum time period during which a secondary-level replica may remain disconnected, before being excluded from the replica group.
1.2) if $V_i < V_j$, then the primitive send returns ERROR, and r_j sends to r_i the current primary-level replicas group. r_i will re-send the update a to r_j. The maximum delay for the reception of a is computed as above.
1.3) if $V_i > V_j$, proceeds as in the case 1.1)

2) Let r_i be a secondary-level replica, which calls send(a).
2.1) if $V_i == V_j$, then r_j forwards a to all primary-level peers and to the other secondary-level replicas. The maximum delay for a to be received by the primary-level replicas is null. The maximum delay for a to be received by the secondary-level replicas is computed as in the case 1.1).
2.2) if $V_i < V_j$, proceeds as in the case 1.2)
2.3) if $V_i > V_j$, proceeds as in the case 2.1)

10.2 Resolving the other replication decisions

Another subject for future work concerns the resolution of the other replication decisions, including: replica creation, replica placement, replica migration and replica deletion. We sketch the approach, which we envisage, for integrating these decisions within our Replicated Service Hosting System.

10.2.1 The placement scope

We define the *placement scope* as the group of machines, which candidate for hosting the replicas, that will be created.

The algorithm for creating the *placement scope* (Figure 10.2) combines the distribution of client bindings with the performance contract, as follows. Firstly, it obtains from the local Replica Repository, the distribution of client bindings per domains. Secondly, for each domain D_i, originating client requests, it computes the *access frequency*, as the ratio between the number of client bindings to replicas in that domain and the number of replicas available in that domain.

```
create(service[, scope]) {
  if (scope is null)
    scope = getPlacementScope(service);
  hosts = getHosts(scope);
  while (hosts is null) and (scope is not null) {
    new_scope = getPlacementScope(service, scope);
    if (new_scope)
     hosts = getHosts(new_scope);
    scope = new_scope;
  }
  if (hosts is null) return null;
  criterion=getPlacementCriterion(service);
  selected = rank(hosts, criterion);
  if (selected)
    place(service, selected);
  return selected;
}

place(service, host) {
  instantiate service on host
  replica = createReference(host);
  add(service, replica);
  return replica;
}

delete(service, replica) {
  stops service instance from the replica's host
  delete(service, replica);
}

migrate(service, replica[, outside]) {
  if (outside)
   create(service);
  else
   create(service, crt_domain);
  delete(service, replica)
}
```

Figure 10.3: The other replication base primitives

Next, it computes for each domain, the product, noted *load_ domain*, between the *access frequency* and the percentage of unsatisfied requests, which occurred since the last replica creation (also obtained by querying the local Replica Repository). The *placement scope* is chosen as the domain for which *load_ domain* has the largest value. An optional parameter contains a list of domains that must be ignored when looking for the domain with the largest value of *load_ domain*.

10.2.2 The replication primitives

The protocols resolving the creation, the placement, the deletion and the migration decisions are encapsulated within the primitives create, place, delete and migrate (showed in Figure 10.3).

The primitive create resolves the creation decision. It takes a mandatory argument, which is the service name and an optional argument which specifies the list of domains, where the new replica should be created. If this is empty, then the new replica will be created in the placement scope (computed by the algorithm in Figure 10.2). If the placement scope doesn't contain any available hosts, then it is augmented incrementally with new domains, until at least an available host is found. The new replica host is determined, by ranking the available hosts by their values obtained for the metric contained in the placement criterion. If a suitable host has been found, then the primitive place is invoked, in order to resolve the placement decision.

The primitive place creates a new service instance on the chosen host, and inserts its reference into the replica repository.

The primitive delete resolves the deletion decision, by stopping the local service instance and by deleting its reference from the replica repository.

The primitive migrate resolves the migration decision. This primitive has two mandatory parameters, which are the service name and the replica that should be moved. Optionally, the primitive takes a third parameter, which has two possible options: *inside the domain*, *outside the domain*. The former option requires that the replica should be moved in the current domain. This option is pertinent when the number of the replicas in the current domain is below the admitted replication degree. In this case, the host of the new replica is chosen among the hosts available in the current domain. The latter option is pertinent when the maximum replication degree within the current domain has been exceeded. In this case, the replica should be moved in a different domain within the placement scope. In both cases, the primitive migrate invokes the primitive create, followed by the primitive delete.

Each Replica Manager instance controls the evolution of the replicas which belong to the domain, to which it is attached. Precisely, it places replicas only in its domain, it migrates overloaded replicas and deletes unuseful replicas only from its domain. In other words, the methods place, migrate and delete can be invoked only by the Replica Manager instance attached to the replica's domain.

The creation and migration decisions claims for the consistency of the bindings repositories, maintained by the Replica Managers, located in different domains. As the eventual consistency of the bindings repositories suffices, each Replica Manager should propagate periodically to its peers, the updates that it performed to its local repository. Also, no concurrency control is needed, because the updates are commutative and non-conflicting.

10.2.3 Predefined policies for replica creation decision

We configure the replica creation decision by two main policies, defined as follows: 1) *if y false positive reactions occur during a given time period, then check the creation criterion, in order to determine if a new replica is needed*;
2) *if z no adaptation reactions occur during a given time period, then check the creation criterion, in order to determine if a new replica is needed*;

These policies aim to limit the number of reactions of type *false positive*, respectively, of type *no adaptation*. The conditions formulated within these policies indicate that there is a global problem, that should be further investigated by checking the creation criterion, in order to determine if the addition of a new replica could improve the clients' perceived performance.

Other perspectives for continuing this work include:

1. inferring automatically the service workload;
2. detailing what are the valid combinations of consistency options (with respect to the service semantics and to the execution environments);
3. checking automatically if a particular consistency contract is valid (and rejecting it, if this is not the case);
4. providing the adaptation feature, so as to adapt the consistency options according to the environment conditions;
5. comparing experimentally various response time estimators, by varying the actual system and network conditions;
6. providing estimators for other service-specific metrics, like the data throughput and the availability;
7. studying experimentally when replica creations and deletions should be performed, at run-time;
8. formalize the performance guarantees that the system is able to provide (with a given probability);

Bibliography

[1] F. Akal, C. Turker, H.S. Schek, Y. Breitbart, T. Grabs, and L. Veen. Fine-Grained Replication and Scheduling with Freshness and Correctness Guarantees. In *Proceedings of the 31nd International Conference on Very Large Data Bases (VLDB)*, Trondheim, Norway, 2005.

[2] R. Alonso, D. Barbara, and H. Garcia-Molina. Data Caching Issues in an Information Retrieval System. *ACM Transactions on Database Systems*, 15(3):359– 384, 1990.

[3] B. Badrinath and K. Ramamritham. Semantics-Based Concurrency Control: Beyond Commutativity. *ACM Transactions on Database Systems*, 17(1):163–199, 1992.

[4] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Jose, California, USA, 1995.

[5] G. Brun-Cottan and M. Makpangou. Adaptable Replicated Objects in Distributed Environments. Technical Report RR-2593, Inria, May 1995.

[6] V. Cardellini, M. Colajanni, and P. S. Yu. Geographic Load Balancing for Scalable Distributed Web Systems. In *Proceedings of the 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2000.

[7] R. Carter. *Performance Measurement and Prediction in Packet-Switched Networks: Techniques and Applications.* PhD thesis, Boston University, Boston, MA, 1997.

[8] I. Chabbouh and M. Makpangou. Caching Dynamic Content with Automatic Fragmentation. In *Proceedings of the 7th International Conference on Information Integration and Web Based Applications & Services (iiWAS)*, Kuala Lumpur, Malaysia, October 2005.

[9] M. Chen and W. Mao. Anycast By DNS Over Pure IPv6 Network. Department of Electrical Engineering and Computer Science, University of California, Berkeley, 2001.

[10] G. Chockler, D. Dolev, R. Friedman, and R. Vitenberg. Implementing a Caching Service for Distributed CORBA Objects. In *Proceedings of the IFIP/ACM Middleware'00*, 2000.

[11] Cisco. Cisco Content Routing Protocols. In *white paper*, March 2001.

[12] L. Cox and B. Noble. Fast Reconciliations in Fluid Replication. In *Proceedings of the 21th IEEE International Conference on Distributed Computing Systems (ICDCS)*, Phoenix, Arizona, USA, April 2001.

[13] M. Crovella and R. Carter. Dynamic Server Selection Using Bandwidth Probing in Wide-Area Networks. In *Proceedings of the Conference on Computer Communications (INFOCOM)*, 1997.

[14] K. Daudjee and K. Salem. Lazy Database Replication with Snapshot Isolation. In *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB)*, Seoul, Korea, 2006.

[15] Dilley, B. Maggs, J. Parikh, H. Prokop, R. Sitaraman, and B. Weihl. Globally Distributed Content Delivery. *IEEE Internet Computing*, 6(5):50–58, 2002.

[16] Z. Fei, M. Ammar, and E. Zegura. Multicast Server Selection: Problems, Complexity and Solutions. *IEEE Journal on Selected Areas in Communication*, 20(7):1399–1413, 2002.

[17] P. Felber and A. Schiper. Optimistic Active Replication. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS)*, Mesa, AZ, USA, 2001.

[18] C. Ferdean and M. Makpangou. A Generic and Flexible Model for Replica Consistency Management. In *Proceedings of the International Conference on Distributed Computing & Internet Technology (ICDCIT)*, Bhubaneswar, India, 2004.

[19] C. Ferdean and M. Makpangou. Exploiting Application Workload Characteristics to Accurately Estimate Replica Server Response Time. In *Proceedings of the Distributed Objects and Applications (DOA)*, Agia Napa, Cyprus, October 2005.

[20] C. Ferdean and M. Makpangou. A Fine-Grained Customizable Consistency Protocol for Replicated Data Objects. In *Proceedings of the Journées Francophones sur la Coherénce en Univers Réparti (CDUR)*, Paris, France, November 2005. CNAM.

[21] C. Ferdean and M. Makpangou. A Response Time-Driven Server Selection Substrate for Application Replica Hosting Systems. In *Proceedings of the Symposium on Applications and Internet (SAINT)*, Phoenix, Arizona, USA, January 2006.

[22] S. Gancarski, H. Naacke, E. Pacitti, and P. Valduriez. The Leganet System: Freshness-Aware Transaction Routing in a Database Cluster. *Information Systems Journal*, Elsevier, 2006.

[23] N. Gibelin and M. Makpangou. Efficient and Transparent Web-Services Selection. In *Proceedings of the International Conference on Service-Oriented Computing (ICSOC)*, Amsterdam, Netherlands, 2005.

[24] Statistics Glossary. http://www.stats.gla.ac.uk/steps/glossary/presenting_data.html.

[25] R. A. Golding and D. E. Long. The Performance of Weak-Consistency Replication Protocols. Technical Report UCSCCRL-92-30, University of California at Santa Cruz, July. 1992.

[26] J. Guyton and M. Schwartz. Locating Nearby Copies of Replicated Internet Servers. In *Proceedings of the Special Interest Group on Data Communication (SIGCOMM)*, Cambridge, MA, August 1995.

[27] J. Gwertzman and M. Seltzer. The Case for Geographical Push-Cashing. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems (HotOS)*, Orcas Island, WA, USA, May 1995.

[28] A. Jebali and M. Makpangou. Replica Divergence Control Protocol in Weakly Connected Environment. In *Proceedings of the IEEE International Symposium on Network Computing and Applications*, Cambridge, MA, USA, 2001.

[29] A.M. Kermarrec, I. Kuz, M. Van Steen, and A. S. Tanenbaum. A Framework for Consistent, Replicated Web Objects. In *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems (ICDCS)*, Amsterdam, Netherlands, May 1998.

[30] A.M. Kermarrec, A. Rowstron, M. Shapiro, and P. Druschel. The IceCube Approach to the Reconciliation of Divergent Replicas. In *Proceedings of the 20th ACM Symposium on Principles of Distributed Computing (PODC)*, Newport, RI, USA, August 2001.

[31] J.J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, 10(1), 1992.

[32] N. Krishnakumar and A.J. Bernstein. Bounded Ignorance in Replicated Systems. In *Proceedings of the 10th ACM Symposium on Principles of Database Systems*, Denver, Colorado, May 1991.

[33] S. Krishnamurthy, W. H. Sanders, and M. Cukier. An Adaptive Framework for Tunable Consistency and Timeliness Using Replication. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, Bethesda, Maryland, 2002.

[34] R. Ladin, B. Liskov, and L. Shrira. Lazy Replication: Exploiting the Semantics of Distributed Services. In *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing*, Quebec City, CA, August 1990.

[35] V. Martins, E. Pacitti, and P. Valduriez. A Dynamic Distributed Algorithm for Semantic Reconciliation. In *Records of the 7th International Meeting Distributed Data & Structures 6 (WDAS)*, Santa Clara, California, 2006.

[36] Mathworld. http://mathworld.wolfram.com/leastsquaresfitting.html.

[37] D. Mosberger. Memory Consistency Models. Technical report, University of Arizona, November 1993.

[38] Mysql. www.mysql.org.

[39] B. Noble, B. Fleis, and M. Kim. A Case for Fluid Replication. In *Proceedings of the Network Storage Symposium (NetStore)*, Seattle, WA, USA, October 1999.

[40] J. O'Brien and M. Shapiro. An Application Framework for Collaborative, Nomadic Applications. In *Proceedings of the International Conference on Distributed Applications and Interoperable Systems (DAIS)*, Bologna Italy, June 2006.

[41] Chord P2P Overlay. http://pdos.csail.mit.edu/chord.

[42] E. Pacitti, C. Coulon, P. Valduriez, and M.T. Ozsu. Preventive Replication in a Database Cluster. *Distributed and Parallel Databases*, 18(3):223–251, 2005.

[43] E. Pacitti, P. Minet, and E. Simon. Fast Algorithms for Maintaining Replica Consistency in Lazy Master Replicated Databases. In *Proceedings of the 25nd International Conference on Very Large Data Bases (VLDB)*, Edinburgh, Scotland, 1999.

[44] E. Pacitti, E. Simon, and R. Melo. Improving Data Freshness in Lazy Master Schemes. In *Proceedings of the 18nd International Conference on Distributed Computing Systems (ICDCS)*, Amsterdam, Netherlands, 1998.

[45] C. Le Pape, S. Gancarski, and P. Valduriez. REFRESCO: Improving Query Performance through Freshness Control in a Database Cluster. In *Proceedings of the International Conference on Cooperative Information Systems (CoopIS)*, Agia Napa, Cyprus, 2004.

[46] F. Pedone and A. Schiper. Generic Broadcast. In *Proceedings of the 13th International Symposium on Distributed Computing (DISC)*, Bratislava, Slavak Republic, September 1999.

[47] F. Pedone, M. Wiesmann, B. Kemme A. Schiper, and G. Alonso. Understanding Replication in Databases and Distributed Systems. In *Proceedings of the 20th IEEE International Conference on Distributed Computing Systems (ICDCS)*, Taipei, Taiwan, 2000.

[48] K. Peterson, M.J. Spreitzer, D.B. Terry, M.M. Theimer, and A.J. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, Saint Malo, France, October 1997.

[49] G. Pierre, M. Van Steen, and A.S. Tannenbaum. Dynamically Selecting Optimal Distribution Strategies for Web Documents. *IEEE Transactions on Computers*, 51(6):637–651, 2002.

[50] G. Pierre and M. van Steen. Globule: a Collaborative Content Delivery Network. *IEEE Communications Magazine*, 44(8), 2006.

[51] N. Prequica, M. Shapiro, and J. Legatheaux Martins. Automating semantic-based reconciliation for mobile transactions. In *Proceedings of the Conférence Francaise sur les Systèmes d'Exploitation (CFSE)*, La-Colle-sur-Loup, France, October 2003.

[52] C. Pu, W. Hseush, G. E. Kaiser, K.-L. Wu, and P. S. Yu. Distributed Divergence Control for Epsilon Serializability. In *Proceedings of the 13th International Conference on Distributed Computing Systems (ICDCS)*, Pittsburgh, USA, 1993.

[53] M. Rabinovich and A. Aggarwal. Radar: A Scalable Architecture for a Global Web Hosting Service. In *Proceedings of the 8th World Wide Web Conference (WWW8/Computer Networks)*, Toronto, Canada, May 1999.

[54] M. Rabinovich, Z. Xiao, and A. Agrawal. Computing on the Edge: A Platform for Replicating Internet Applications. In *Proceedings of the 8th International Workshop on Web Content Caching and Distribution*, NY, USA, September 2003.

[55] Radware. Web Server Director. In *white paper*, 2002.

[56] P. Reiher, J. Heidemann, D. Ratner, G. Skinner, and G. Popek. Resolving File Conflicts in the Ficus File System. In *Proceedings of the USENIX Summer Conference*, Boston, MA, USA, June 1994.

[57] G. Ricart and A.K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, 1981.

[58] L. Rilling, S. Sivasubramanian, and G. Pierre. High Availability and Scalability Support for Web Applications. In *Proceedings of the IEEE International Symposium on Applications and the Internet*, January 2007.

[59] M. Sayal, Y. Breitbart, P. Scheuermann, and R. Vingralek. Selection Algorithms for Replicated Web Servers. In *Proceedings of the Workshop on Internet Server Performance (WISP/SIGMETRICS)*, Madison, WI, June 1998.

[60] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. Department of Computer Science, Cornell University, Ithaca, New York 14853.

[61] M. Shapiro. *La gestion des objets dans les systèmes répartis de large échelle, Habilitation à diriger les recherches.* PhD thesis, Université Paris VI - Pierre et Marie Curie, Paris, France, 2002.

[62] M. Shapiro and N. Krishna. The three dimensions of consistency. In *Proceedings of the Journées Francophones sur la Coherénce en Univers Réparti (CDUR)*, Paris, France, November 2005. CNAM.

[63] A. Singla, U. Ramachandran, and J. Hodgins. Temporal Notions of Synchronization and Consistency in Beehive. In *Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures*, Newport, RI, June 1997.

[64] S. Sivasubramanian, G. Alonso, G. Piere, and M. van Steen. GlobeDB: Autonomic Data Replication for Web Applications. In *Proceedings of the 14th International World-Wide Web Conference*, Chiba, Japan, May 2005.

[65] S. Sivasubramanian, M. Szymaniak, G. Pierre, and M. van Steen. Replication for Web Hosting Systems. *ACM Computing Surveys*, 36(3):291–334, 2004.

[66] R. Strom, G. Banavar, K. Miller, A. Prakash, and M. Ward. Concurrency Control and View Notification Algorithms for Collaborative Replicated Objects. *IEEE Transactions on Computers*, 47(4):458–471, 1998.

[67] S. Susarla and J. Carter. Khazana: A Flexible Wide Area Data Store. Technical Report UUCS-03-020, School of Computing, University of Utah, Salt Lake City, October 2003.

[68] Network Weather System. http://nws.cs.ucsb.edu/.

[69] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. Session Guarantees for Weakly Consistent Replicated Data. In *Proceedings of the third IEEE International Conference on Parallel and Distributed Information Systems (PDIS)*, Austin, Texas, USA, September 1994.

[70] D. B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the 15th ACM Symposium on the Operating Systems Principles (SOSP)*, Copper Mountain Resort, Colorado, December 1995.

[71] F. Torres-Rojas, M. Ahamad, and M. Raynal. Timed Consistency for Shared Distributed Objects. In *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing (PODC)*, Atlanta, May 1999.

[72] TPC-W. www.tpc.org/tpcw.

[73] A. Vahdat, P. Eastham, and T. Anderson. WebFS: A Global Cache Coherent File System. University of California, Berkeley, 1996. Department of Computer Science.

[74] R. Vingralek, Y. Breitbart, M. Sayal, and P. Scheuermann. Web++: A System For Fast and Reliable Web Service. In *Proceedings of the USENIX Annual Technical Conference*, Sydney, Australia, June 1999.

[75] R. West, K. Schwan, I. Tacic, and M. Ahamad. Exploiting Temporal and Spatial Constraints on Distributed Shared Objects. In *Proceedings of the 17th IEEE International Conference on Distributed Computing Systems (ICDCS)*, 1997.

[76] H. Yu and A. Vahdat. Design and Evaluation of a Continuous Consistency Model for Replicated Services. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, USA, October 2000.

[77] E. Zegura, M. Ammar, Z. Fei, and S. Bhattacharjee. Application-Layer Anycasting: A Server Selection Architecture and Use in a Replicated Web Service. *IEEE/ACM Transactions on Networking*, 8(4):455–466, August.

[78] C. Zhang. Consistency and Replication. Miami, FL, 2005. School of Computing and Information Sciences.