Sabbarus: Distributed plate-form for replica divergence control, design and implementation

Ahmed Jebali

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Sabbarus : *Distributed plate-form for replica divergence control, design and implementation*

Ahmed Jebali

N° 5356

October 2004

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--5356--FR+ENG

Sabbarus : Distributed plate-form for replica divergence control, design and implementation

Ahmed Jebali*

Thème COM — Systèmes communicants
Projets REGAL

Rapport de recherche n° 5356 — October 2004 — 20 pages

Abstract: For many distributed application, it is still hard to make correct decision while using not up to date knowledge of the system state (network latency, failure...). In load balancing system, there is a difficulty to learn at all instant the system load to assign a processor to a submitted task. For such applications, if the overall system state is known with a certain limited error the distributed decision will be correct. We target applications formed of processes located world-wide. These processes cooperate by sharing a replicated object that represents the system state. The shared object is replicated on each process. This paper describes Sabbarus a distributed plate-form offering replicas divergence control protocol for these applications. We consider processes with well-known profiles; however the actual behaviour of each process could diverge from the profile.

Key-words: Distributed System, Network Service, Replication Protocol, Divergence Control, Bounded Consistency, Replication Plate-form.

* ahmed.jebali@inria.fr

Sabbarus : **Plate-forme distribué de contrôle de divergence, conception et implantation**

Résumé : Pour de nombreuses applications réparties il est difficile de faire une décision distribuée si on connaît pas l'état du système (latence réseau, fautes de communication...). Par exemple, dans un système d'équilibrage de charge distribué, l'affectation d'une tâche dépend de la charge de chaque processeur. Pour ce type d'application, si on connaît l'état du système avec une certaine erreur, la décision distribuée serait correcte. Nous ciblons des applications formées par des processus coopérant à large échelle par le partage d'un objet. Cet objet est répliqué sur chaque processus. Ce papier décrit Sabbarus une plate-forme distribuée de contrôle de divergence. Nous considérons des processus avec des profils bien connus. Cependant, le comportement de chaque processus peut diverger de son profil.

Mots-clés : Ssysteme distribué, Service réseau, Protocole de réplication, Contrôle de divergence, Divergence bornée, Plate-forme

1 Introduction

The main issue of writing a distributed application is to handle correctly communication between peers. Therefore using "standard" facilities like sockets or RPCs can be a hard task due to communication problems and network failures. In some distributed applications, data must be replicated to increase application performance and access latency. In a replicated system, managing the coherency of replicas is the main problem.

Sabbarus is a framework that offers a set of functionalities for writing replicated application. Using Sabbarus , avoid to programmers dealing directly with communication problem and network failure. Sabbarus is dedicated for application accepting divergence between replicas. Sabbarus uses a profiling mechanism to correct instant divergence. Profiles are learned by collecting data flow characteristics between applications and Sabbarus replicas managers. The design goals of Sabbarus is use easiness and application independence. After presenting some targeted applications in section 2, we describe briefly the divergence control protocol in section 3. For more details about Sabbarus protocol and its internals, see the documentation in: http://www-sor.inria.fr/projects/Sabbarus/. In section 4 and 5, we detail Sabbarus design and implementation. In section 6, we point out benefits of using Sabbarus and two applications that we developed. We discuss some implementation details in section 7. Finally, we conclude in section 8.

2 Targeted applications

The protocol is designed with certain targeted applications in mind. First, we are targeting cooperative applications made of groups of cooperative processes deployed on geographically separate sites. We anticipate that the common state shared by these processes could be encapsulated by a well defined abstract object. Each cooperating process holds a local copy of the shared object. The shared object is fully determined by its initial state and the sequence of operations that application processes initiate on it; consequently re-executing the same operation sequence from the same initial state will produce an identical object state. Operations initiated on the object have two properties: (1) measurable, i.e., one can define for each operation a numerical value that indicates its effect on the shared object and (2) predictable, i.e., one can extract from application historic an evolution model using any existing monitoring and statistics techniques. Historic analysis can be based on traces formerly collected or on "in fly" traffic capture. Finally targeted applications correctness is not strongly related to the accuracy of replicas states, thus we exclude applications relying on strong consistency guarantees.

2.1 Airline reservation

An air reservation system deals with flight tickets. the shared object for concurrent seller travel agencies is the number of available seats on a flight. The measure of a reservation operation can be in a basic way -1 or +1 if the reservation is canceled or new seats are

added. Travel agency can see non coherent values of the number of available seats, this can correspond to over-booking practice for example.

2.2 Load balancing

A load balancing system is a front-end between job submitters and execution system. A job owner submits the task to a local load balancing manager, that is responsible for scheduling it in such a way that the overall load is equally distributed between execution processors. For this example, the measure of a job submit operation can be the estimated resources to be used by the job while running. Load balancing managers share the values of real load on each processor.

2.3 Statistics collectors and analyzers

Nowadays, there is an increasing interest to statistical collectors and analyzers. Those systems collect continuous information flow and share it for visualization purpose like client flow in large commercial surfaces or car circulation in a town center. Collecting and analyzing this data allow making fine prediction and regulation. This application will benefit from our model, if given a correct representation of statistical measurements.

3 Divergence Control protocol

3.1 Replication model

We consider distributed applications composed of groups of replicated processes noted $(p_1, \ldots, p_n)$ and located on different sites noted $(S_1, \ldots, S_n)$. These processes cooperate through a shared replicated object. There is one replica of the shared object on each site. Each site is associated with a *consistency manager* that controls operations performed on the local replica of the shared object. An *operation* is an event chain that leads to a particular access to the shared object.

Each process accesses the shared replicated object by initiating operations, via the local consistency manager. Each executed operation by one process is eventually propagated to all consistency managers attached to replicas of the shared object. Each manager decides the order in which its associated replica of the shared object performs operations initiated both locally and by remote process. The primary role of a consistency manager is to make sure that new operations can be executed by the local process without compromising application consistency. For that, each consistency manager is parameterized with an applicative consistency criterion that is evaluated when the local process wants to initiate a new operation. If the evaluation succeeds, the requested operation is accepted and can therefore be scheduled on this site regardless the activity on other sites. Otherwise, a synchronization with other consistency managers is required.

The synchronization allows consistency managers to reduce the divergence between their local views of the application's state and the real one (that is the set of operations initiated

so far by the group of processes composing the application). Note that the model doesn't specify the way operations initiated on one site happen to be seen by the consistency manager located on a different site, nor the acknowledgement mechanism. One can use the anti-entropy protocol [PST+97], gossip messages [LLSG92], or any reliable propagation protocol. A more detailed description of the replication model can be found in [JM01, JM02, BC98].

3.2 Bounding Replica Divergence

3.2.1 Divergence Definition

A replica is fully determined by the sequence of operations that it executes. The shared object is the real object obtained by executing all operations initiated on all replicas. Thus the divergence is the difference between the replica and the real shared object. This difference depends on the operations locally accepted on replicas and not propagated to others. The difficulty is how to determine these operations without continuously communicating with other replicas.

Let us assume that the sequence of operations which a process p_i is likely to initiate during the application's lifetime is characterized by a profile function denoted $\mathcal{P}_i$. The profile maps to each time interval the history of operations that are expected from p_i during that time interval [1].

The idea is to allow consistency managers to use profile functions to approximate remote process histories, without having to communicate with them, while still guaranteeing application consistency. Since prediction is not always accurate, these approximations could diverge from the actual behaviours of processes. The more accurate is the prediction provided by the profile functions, the smaller is that divergence. Our goal is to provide consistency managers with a way to bound this divergence to a value B.

3.2.2 Bounding the Divergence

We note $I_i(t', t) = (op_1, op_2, \ldots, op_k)$ the ordered sequence of operations initiated by process p_i during the time interval $[t', t]$. We simply note $I_i(t)$ if t' is the start time of the applications. With respect to a process $p_j, j \neq i$, the history of p_i is composed of two parts: (1) $A_{i,j}$ the sequence of operations initiated by p_i and already acknowledged by site S_j and (2) $W_{i,j}$ the sequence of operations initiated by p_i yet to be propagated to the consistency manager on site S_j or yet to be acknowledged by that consistency manager. Using a concatenation operator $\oplus$ [2] on histories, we have the following equality: $I_i(t) = A_{i,j}(t') \oplus W_{i,j}(t', t)$.

We attach to an operation op a measure $\mu(op)$ that represents the impact of its application on the shared object. For example, the measure of $get(amount)$ on a stock is $(-amount)$ since its impact is to decrement the stock by the value $amount$. The measure of a history is

[1] We will see later that only the impact of predicted operations is used in the protocol.

[2] The $\oplus$ operator concatenates histories. $h_1 \oplus h_2$ is the history formed by operations of h_1 followed by operations of h_2.

the sum of measures of operations of this history. For a given history $\mathcal{H} = (op_1, op_2, \ldots, op_k)$ the measure is: $\mathcal{M}(\mathcal{H}) = \mu(op_1) + \mu(op_2) + \ldots + \mu(op_k)$.

Consider now the tuple RH that represents the real histories that actually occurred on processes $(p_1, \ldots, p_n)$;

$$RH(t) = (I_1(t), I_2(t), \ldots I_n(t))$$

A consistency manager on site S_i approximates this global state with EH_i as follows:

$$EH_i(t) = (A_1(T_i[1]) \oplus \mathcal{P}_1(T_i[1], t), \ldots, I_i(t), \ldots,$$

$$A_n(T_i[n]) \oplus \mathcal{P}_n(T_i[n], t))$$

where $T_i[j]_{1 \leq j \leq n}$ is the occurrence date of the last operation initiated by process p_j and already acknowledged by p_i; $\mathcal{P}_j$ is the profile function of process p_j. We define the divergence (d_i) between RH and EH_i as follows:

$$d_i = |\mathcal{M}(I_1(t), I_2(t), \ldots I_n(t)) - \mathcal{M}(A_1(T_i[1])$$

$$\oplus \mathcal{P}_1(T_i[1], t), \ldots, I_i(t), \ldots, A_n(T_i[n]) \oplus \mathcal{P}_n(T_i[n], t))|$$

rewritten to:

$$d_i = |\Sigma_{j \neq i}^n \mathcal{M}(I_j(t)) - \mathcal{M}(A_j(T_i[j]) \oplus \mathcal{P}_j(T_i[j], t))|$$

Since $I_j(t) = A_j(T_i[j]) \oplus W_j(T_i[j], t)$, the divergence is:

$$d_i = |\Sigma_{j \neq i}^n \mathcal{M}(W_j(T_i[j], t) - \mathcal{M}(\mathcal{P}_j(T_i[j], t))|$$

Using the properties of absolute values, the present equation implies that:

$$d_i \leq \Sigma_{j \neq i}^n |\mathcal{M}(W_j(T_i[j], t) - \mathcal{M}(\mathcal{P}_j(T_i[j], t))|$$

Hence, to guarantee that d_i is less than a fixed bound B, it suffices to find a tuple local bounds $(b_1, \ldots, b_n)$ such that:

$$\forall j,\ b_j \geq 0 \ and \ \Sigma_{j=1}^n b_j \leq B \tag{1}$$

$$\forall j,\ |\mathcal{M}(W_j(T_i[j], t) - \mathcal{M}(\mathcal{P}_j(T_i[j], t))| \leq b_j \tag{2}$$

To guarantee these conditions it suffices that each consistency manager constraints its local behaviour to be conform to its profile. The protocol has just to enforce this local condition and propagates operations to its peers when needed. For a complete study of elementary conditions extraction see [**?**].

3.3 Divergence Control Protocol

3.3.1 Algorithm

The protocol bounds the divergence of the actual history of each process with respect to approximations that are made by other partners. We consider a tuple $(b_i, \ldots, b_n)$ such that $\Sigma_{j=1}^{n} b_j \leq B$. We assign the value b_j to the consistency manager on site S_j. When a process p_j wants to execute an operation op at date t, it informs its consistency manager, which in turn performs the following algorithm:

1. If for all i, the condition $|\mathcal{M}(W_j(T_i[j], t) + \mu(op) - \mathcal{M}(\mathcal{P}_j(T_i[j], t)| \leq b_j$ holds then the operation is accepted and added to the local history.
2. Otherwise, a synchronization is required. The local manager sends to each partner k such that $|\mathcal{M}(W_j(T_k[j], t) + \mu(op) - \mathcal{M}(\mathcal{P}_j(T_k[j], t)| > b_j$, operations initiated since the last operation acknowledged by k.

A process can also send notification at any time; this is not necessary to ensure the divergence condition, but increases system performance. When receiving a notification message from p_j, p_k updates the corresponding entry in its notification date vector T_k and executes notified operations on its own replica.

4 Sabbarus Framework

"Sabbar" is a kind of Cactus. The Sabbarus framework allows replicas to progress without having to communicate all the time, just like cactus plants that can be "disconnected" for a long time from any water source! Sabbarus implements the divergence control protocol 3.3. It implements as well the operation concept, profile building and an administration and control interface.

Sabbarus is written over Ensemble. It performs a simple, and compact interface to write and administrate replicated application. The main functionalities of Sabbarus are:

- a library for developing clients application accessing a replicated object;
- a mechanism for defining specific access to the replicated object;
- a command line interface for controlling the framework.

A synoptic of Sabbarus is shown in Figure 1. Sabbarus is composed of a group of *regulators* in top of a group communication support and access *proxies*. *Regulators* implements the abstraction of *coherency manager* CM. They are hosted on a set of dedicated machines (one *regulator* per machine) over a WAN. They implement the divergence control protocol. Each *Regulator* controls a replica of an application[3]. Replicas are located on the *regulator* hosts. *Regulators* access to replicas by calling exported primitives from a *Replica Access Module*

[3]We consider here the case of one application. The implementation allows to run multiple applications

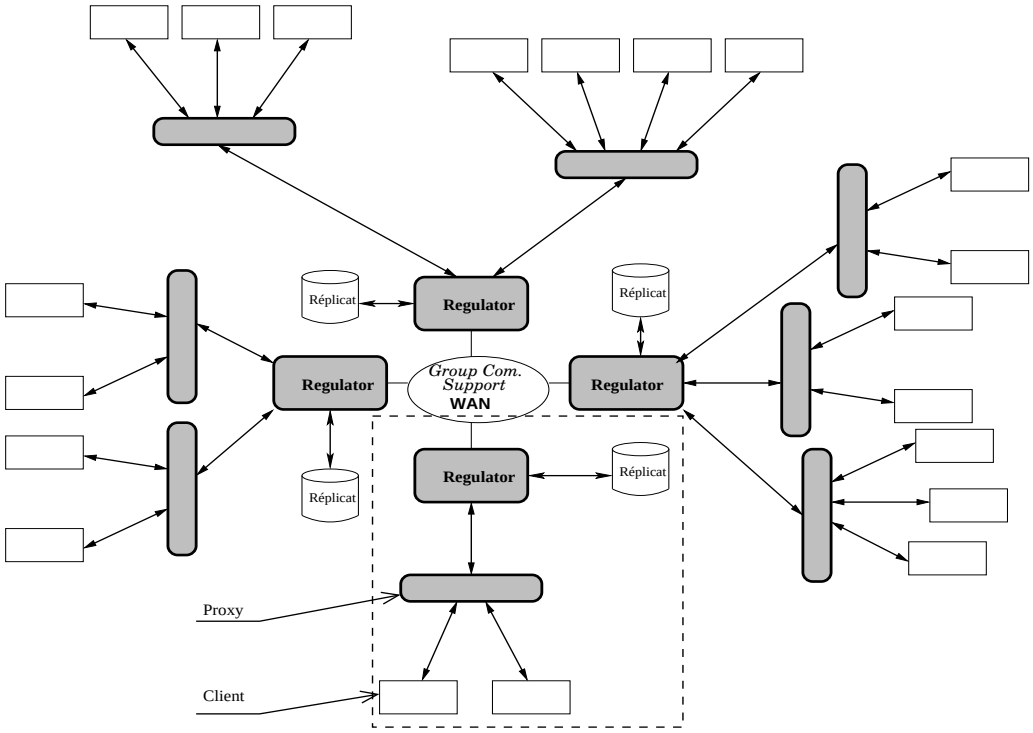

Figure 1: A synoptic of Sabbarus

(Figure 2). Proxies are the Sabbarus front-ends. Each *regulator* serves one or more proxies. Application clients are linked with a Sabbarus Access Module that exports primitives for operations requests. Typically, proxies are deployed over a MAN. Administrator managers the application using control interface. Finally, each proxy implements a profiling interface to communicate with an external profiling tool.

4.1 The Sabbarus Proxy and *Regulator*

The Sabbarus proxy is the front-end of Sabbarus core, clients send requests to the *regulator* and receive respond through the proxy. Proxy handles also communication with administrator and profiling tool. Proxies are daemons that are hosted near the clients. Each proxy is connected to one *regulator*.

Regulators controls access to its replica. *Regulators* implement the divergence control protocol. Main operations of a *regulator* are: serving proxies requests, notification exchange

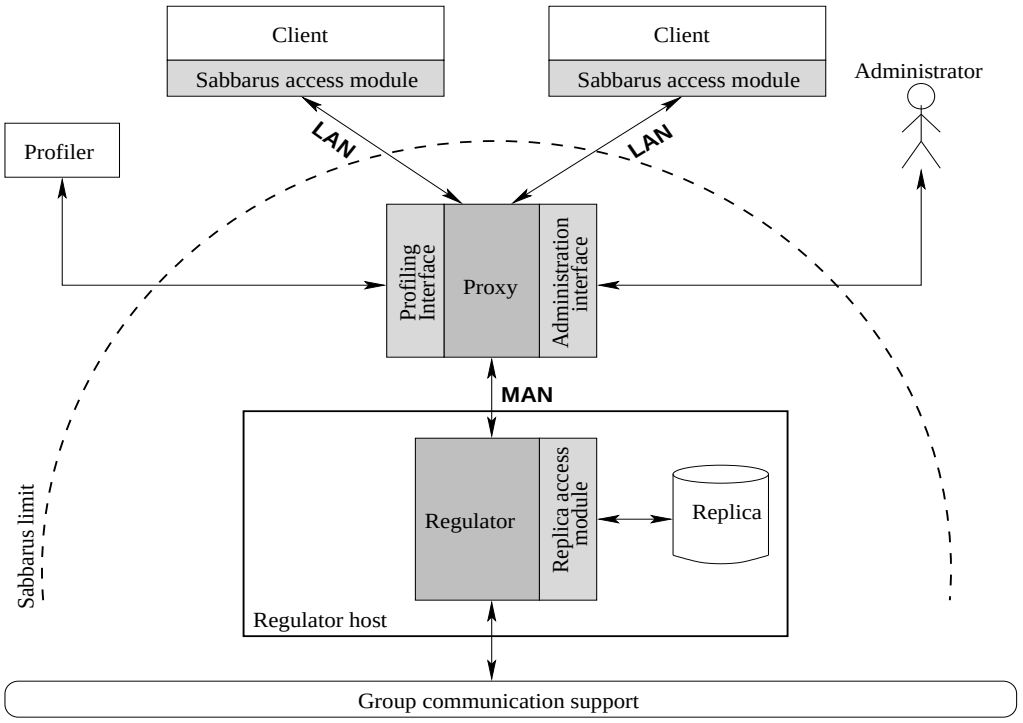

Figure 2: Sabbarus : a *regulator* view.

with others replicas and operation initiation on local replica. Each *regulator* belongs to two groups:

- a local group: composed with its proxies;
- an external group: composed with others *regulators*.

A *regulator* is a daemon hosted on a dedicated machine. It is an event program: it reacts to the events that it receives. An event is a message received. This reaction is implemented by different functions. The *regulator* receives two types of events: events coming from proxies (requests) and events coming from others *regulator*s (notifications).

Regulators communicates over a group communication plate-form. This choice avoid us to write communication primitives — and to choose convenient multicast properties — and allows to take advantage of the membership support.

4.2 Exported Modules

Sabbarus exports two access modules. *Sabbarus access module* is a library that defines the from of request and response fields and functions to send request and receive response. This library is also responsible of an internal marshaling task between external representation of this fields (application client) and internal one (Sabbarus). This module allows to write application with a freely chosen programming language.

Replica access module is used by *regulator* to access to its local replica. An access is only allowed when the request is accepted (e.i. with respect to the divergence control protocol). Access Module is specified by the application programmer with a URL location and an access path to the dependencies. When initializing, *regulator* loads these dependencies and the access module.

4.3 Control and Profiling

The proxy has an administration interface. It is a line command interface that permits a fine control of the Sabbarus platform. Controls include managing the proxy capabilities (number of supported clients, number of requests, scheduling strategies) and regulating the protocol (setting global/local bounds, synchronizing, flushing and purging requests, setting profiles).

The proxy has also another interface with an external module called *Profiler*. The *profiler* builds replicas profiles by analyzing the traffic between clients and proxies. A new profile is build each time the administrator asks for it, or when *regulators* has to update their profiles. The *profiler* intercepts the traffic between clients and proxies. Precisely, it monitors communication ports of the proxies. Locating the *profiler* on this link allows us to collect Sabbarus messages on IP links. Compared to group communication message decoding this choice makes easy message decoding. Moreover, for these links the application could benefits from existing monitoring tools to which one can integrates the *profiler*.

Profiles are build each time the administrator requires it or when *regulators* must update it. The profile is build while running the application. Profiles maintain numeric series of

couples: *(operation measure, occurrence date)*. Then it applies a prediction algorithm to extract profiles. We apply two algorithm:

- Linear: this is the most simple technique. For the observations values (x_i, t_i), where x_i is the measure of request i and t_i its occurrence date. The *profiler* accumulates the values x_i within an observation interval $I = [t, t']$ and computes the frequency $f = \frac{\sum_{t_i \in I} x_i}{t'-t}$. The forecast value x_k at date t_k is deduced by $x_k = f \times (t_k - t')$;
- Polynomial: this is a more complex and refined algorithm. The profile is generated by a polynomial [GS84, HTF01]. Forecast value x_k is deduced by : $x_k = \sum_{i=1}^{k-1} \alpha_i x_i^{p_i}$. The parameters α_i and p_i are determined by the algorithm.

5 Implementation

Sabbarus is written in Objective Caml in top of Ensemble group communication plate-form [Hay98]. Ensemble provides a set of protocol layers that are easy composable (with a composition language) to form communication stacks. *regulator* synchronizations are accomplished by immediate asynchronous call backs provided by Ensemble. We examine hereafter how we implement each Sabbarus component.

5.1 Regulators

The *regulator* group forms the core of Sabbarus . *Regulators* implement the replica divergence protocol as described in 3.3. Each *regulator* implements two interfaces, one to communicate with the local proxies and one to communicate with the others *regulators*. Communication interfaces define a set of events handlers.

The local interface enables application processes, sharing the replicated object, to declare requests that they want to perform. The main handler of this interface is requestHandler (see Figure 3). When receiving a request from the proxy, this handler is invoked. *Regulators* define two accumulators effectiveEffect and predictedEffect that correspond respectively to $\mathcal{M}(W_j(T_i[j], t)$ the measure of operation locally initiated and not known by others *regulators* and $\mathcal{M}(\mathcal{P}_j(T_i[j], t))$ the predicted operations measure. Upon the reception of a client request the *regulator* computes the error that may be introduced by this request. The request is accepted only if this error does not exceed the local bound.

When the divergence condition does not hold, the request is rejected[4] and a notification is requested with an immediate callback from *regulator* interface. This second interface handles communications with others *regulators*. Figure 4 gives a list of its most important handlers.

When a *regulator* receives a message from its peers it calls the receive function. Handlers are called depending on the message type. The serverReadyHandler is called to initialize the *regulator* and instantiate the local replica (load *replica access module* and create the

[4]It is also possible to delay the operation execution after synchronization

```
let requestHandler origin properties =
let predictedEffect = (*calls profiles functions*)
 let effectiveEffect = (*calls a cumulative variable*)
 let error = abs (effectiveEffect + properties.measure - predictedEffect)
 if error < b (*the local bound*) then
   (*operation is accepted and executed locally*)
 else
   serverAsync; (*demands an immediate asynchronous callback to send a notification*)
Send (origin, Respond reqNum) (*application response*)
```

Figure 3: Local interface

```
let receive origin msg =
match msg with
 | ServerReady -> serverReadyHandler()
 | Notification -> notificationHandler()
let actionHandler
let heartBeatHandler
```

Figure 4: *Regulator* interface

replica). The notificationHandler is called when a notification is received. A notification includes the list of operations performed by the sender *regulator* and not yet acknowledged by others *regulators*. For a resource allocation application, this list is reduced to a single integer. When a *regulator* receives a notification it adds these notified operations to its local history (i.e. executes notified operations). The actionHandler is called when a member is joining or leaving the group of *regulators*; this handler is responsible of redistributing the global consistency bound B and reinitializing data structures to match the new group view. Finally the heartBeatHandler implements the notification sending function requested by the local interface.

Regulators communicate using an Ensemble stack [Hay98]. We use a reliable group communication stack composed of a virtual synchrony layer, an xfer layer and a local delivery layer. The xfer layer serves to handle safely the view changes when a *regulator* is joining or leaving. We use the local delivery layer as an acknowledgment mechanism. Consequently, a message cast to the group and locally delivered is considered as acknowledged.

```
let initbind ipadr port num = (*Init the communication with the proxy*)
let request op = (*Main call: request an operation from the application proxy*)
let response () = (*A non blocking operation to receive response*)
```

Figure 5: Local interface

5.2 Proxy and access module

The API used is in the Sabbarus access module is shown in Figure 5. It is a library that defines the form of a request and a response fields, and the function to send request and receive response. This library is also responsible of an internal marshaling task between external representation of fields for client programming languages. Only Objective Caml and C are supported. The client API is presented in Figure 5. The initbind function must be called the first time to connect to Sabbarus . It is parametrized with the IP address and the proxy port number. One can think about a dynamic configuration of these parameters with regards to the proxy load or location, but this is out of the scope of the paper. The request op is the main request call where the parameter op is a predefined structure in the access module (see Figure 6). The basic fields of a request are: name, type, number and measure. There is also further used fields. Hint fields is a list of hints given by the client to help handling the request. Hint can be one of the following: Commutative (the request can be scheduled independently from others handled by the proxy), NoProfile (the request is executed without divergence checking, in the case of reading operations), SyncPoint (the operation causes a synchronization when proceeded). The requestClientParam is an application specific field. The application marshals it in the parameters transferred via Sabbarus to the replicas access module. It can be a simple integer or a more complex structure marshaled into a string. Furthermore, the application can use its own algorithm to encrypt and decrypt this field for any confidentiality reasons. This does not affect the Sabbarus protocol but only access performance to the replica.

5.3 Replica access module

The *replica access module* is used by the *regulator* to access to its local replica. An access to the replica is performed only when a request is accepted by the *regulator*. The access module is specified by the application as a location and a path of a dynamically loadable module, with eventually all module dependencies. An access module is loaded by the *regulator* when deploying a new replicated application [5]. Current Sabbarus version supports Ocaml byte code modules. After loading the access module, the *regulator* calls the function initReplica to

[5] First, dependencies are resolved before loading the access module. This include giving access grants to standard libraries and any other modules to the access module.

```
(*Hints on a request*)
type requestClientHint =
NoHint
| Commutative
| SynchPoint
(*Main request type between application and proxy*)
type requestClientProperties = {
(*Part used by Sabbarus*)
mutable requestClientName : string; (*the name of the request*)
mutable requestClientType : requestClientType; (*Request Type*)
mutable requestClientProfile : bool; (*Flag to use or not profiles*)
mutable requestClientNum : int; (*Request sequential number*)
mutable requestClientMeasure : int; (*Request Measure*)
mutable requestClientHint : requestClientHint list; (*Hint on request execution*)
(*Part used by the application*)
mutable requestClientParam : string (*Marshalled parameters from application*)
}
```

Figure 6: Request structure

```
(*Type of action on replica*)
type actionType = {
actionStatus : int;
actionValues : string};;
(*Returned result type*)
type resultType = {
mutable resStatus : int;
mutable resValues : string
}
(*Initiating replica structure*)
let initReplica () = () ;;
let actionReplica (at : actionType) = ({resStatus=0; resValues=""});;
```

Figure 7: *Replica access module* interface

instantiate and initialize the replica. In instance, the application programmer can execute within this function a data base creation, a replica copy or variable initializations. To execute an action on the replica the *regulator* call the actionReplica with the parameter of the corresponding request marshaled by the application. The return value is an integer status for Sabbarus internal use and the application specific return value which is transmitted by the client application via the proxy.

5.4 Profiler

Prediction algorithm are adapted from the TISEAN tool [HKS00]. TISEAN is a tool written in C and Fortran. It is a library of utilities for numerical series analysis. It is organized as separate executable programs. The common input of these programs is a text file of numerical series. We rewrite prediction algorithm as external libraries of the *profiler*. The *profiler* is implemented as a component of Pandora which is a general purpose monitoring platform [PM00]. Pandora provides a support to write monitoring stacks using a set of components. Components are a sort of filters. Each one handles events generated by the precedent component. We used IP stack monitoring to build the *profiler*. Precisely, the *profiler* is the last component of a Pandora stack monitoring traffic of the Sabbarus proxy port. Because of the extensibility of Pandora and the modularity of the *profiler*, one can use any other prediction algorithm or define other monitoring methods.

6 Application design with Sabbarus

6.1 Sabbarus Characteristics

Sabbarus is an easy to use replication plateform. It is characterized by:

- Non direct dependency of the application: application communicates with Sabbarus using the proxy in a client/server mode. This allows using a variety of programming language. Replica access is performed by an application specific module. Sabbarus only loads this module and call their functions;
- Confidentiality: Since the application marshals operation by it self, it can also add encryption to guarantee confidentiality;
- Profiling tool independence: only a well defined communication interface of the *profiler* is required. Application programmer can use others *profilers*;
- Easy deployment: only *regulators* and proxies must be started. Application communicates with standard sockets. Sabbarus access module can be rewritten for most common programming language (actually we have access modules for C and Objective Caml).

6.2 Application programming

To build an application over Sabbarus these steps must be accomplished:

1. write the *replica access module* library, and export the function described in 5.3;
2. write the client application and link it with the Sabbarus access module library;
3. write the client application[6].

We used Sabbarus to build some applications. The first is a replicated allocation application, it can be run on specific system like reservation (hotel, flight, park...) or system resources allocation (disk, bandwidth, cpu...). The second is a shared database.

6.2.1 Replicated allocation

This is an application based on a shared pool of resources. Shared object represents available resources. Clients send their requests via the Sabbarus proxy. Requests are of three types:

- a resource production: adds resources to shared pool;
- a resources consumption: consumes resources from the shared pool;

[6]There is a difference between adapting an existing centralized application and writing a newly replicated that is aware of the distribution and and the Sabbarus framework. We believe that the process of "distributing" a centralized designed application is not trivial.

```
let theReplica = ref int (*The replica itself*)
let initReplica () (*Replica initialization*)
let actionReplica (at : Bindfile.actionType) -> (*Operation execution*)
Bindfile.resultsType
```

Figure 8: Resource allocation *Replica access module*.

- resource number read: reads the number of resources available in the shared pool.

We suppose that all resources are equivalent, thus shared object is represented by an integer that is equal to the available resources. The profile used is a frequency of consuming and liberating resources. The pool is replicated on each *regulator*. It receives requests from the proxy and executes them locally (with respect to Sabbarus protocol). The divergence condition to maintain is that the difference between allocated resources and available ones must be under a certain bound. Figure 8 shows main functions and values of the *replica access module*.

The main function to execute the request when protocol conditions hold is actionReplica. In the at parameter client application specifies the execution parameters, here the amount of resources produced or requested or the type of operation (read or write). Execution results are returned in a structure of type Bindfile.resultsType. This structure is constructed internally by the client access library code.

6.2.2 Data base access

The second application is a replicated database. We suggest here to maintain an aggregate of the database: in the case of replicated employee data base we bound global working hours to a preset value. The application consists of two components

- Functional: implements common functions of a data base (read, write, add, search...)
- Non functional: dedicated for administrators to control (data consistency, aggregates)

Implementing this application is quiet natural in Sabbarus . Actually, *Regulators* are identified as the non functional part. With respect to the divergence condition to hold, they allow or not the access. *Replica access module* implements the access function. Thus, it can be a DBMS or just a wrapper. In our implementation we used a NDBM data base.

7 Discussion

Sabbarus is a middleware solution for replicating distributed applications. Nowadays a considerably number of middlewares are emerging. Sabbarus use a message oriented asynchronous communication, but not like Sun's Java Message Queue [Sun] where clients can send message even when servers are down. This feature will be considered in future implementation of partitioning management. Using message as communication support needs to handle internally the marshaling and demarshaling. For example inefficient marshaling/demarshaling in Distributed Object Computing (DOC) middlewares like Corba [OMG] can leads to poor performance. The marshaling in Sabbarus has two objectives: (i) make easy the data exchange between clients and proxies and (ii) simplify the profile analyzing of this flow. Operation are considered as first class object. Actually, we use a simple marshaling algorithm, a more elaborated solution like XML can although be used.

8 Conclusion

We presented in this paper Sabbarus; a framework offering replicas divergence control protocol for groups of cooperative processes located world-wide. Processes cooperate by sharing a replicated object. We described a divergence control protocol based on predicted profiles. Sabbarus offers mechanism to: access the plate-form, access replica and administrate the system. We developed also a profiler to perform predictions. We showed how Sabbarus can be used easily to build replicated applications and we described two application that we developed. Because of lack of space, we do not present the plate-form performance. However, performance study in [?] are promising. In the near future, we plan to use Sabbarus as a general replication plate-form. In fact, communication protocol is encapsulated in a separate module and can be easily changed.

Sabbarus distribution

Sabbarus is free downloadable from the url:
http://www-sor.inria.fr/projects/Sabbarus/

References

[BC98] Georges Brun-Cottan. *Cohérence de données répliquées partagées par un groupe de processus coopérant à distance.* Thèse de doctorat, Université Paris 6, Pierre et Marie Curie, Paris (France), September 1998.

[GS84] Graham C. Goodwin and Kwai Sang Sin. *Adaptive filtering prediction and control.* Prentice Hall, 1984.

[Hay98] Mark Hayden. The ensemble system. Technical Report TR98-1662, Cornell University, January 1998.

[HKS00] Rainer Hegger, Holger Kantz, and Thomas Schreiber. Nonlinear time series analysis, December 2000. http://www.mpipks-dresden.mpg.de/~tisean/TISEAN_2.1/index.html.

[HTF01] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of statistical learning : data mining, inference and prediction.* Springer, 2001.

[JM01] Ahmed Jebali and Mesaac Makpangou. Replica divergence control protocol in weakly connected environment. In *The IEEE International Symposium on Network Computing and Applications*, Cambridge, MA, USA, October 2001.

[JM02] Ahmed Jebali and Mesaac Makpangou. Replica divergence control protocol based on predicted profile. In *The International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, USA, June 2002.

[LLSG92] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Database Systems*, 10(4):360–391, 1992.

[OMG] OMG . Object Management Group. http://www.omg.org.

[PM00] Simon Patarin and Mesaac Makpangou. Pandora : A flexible network monitoring platform. In *Proceedings of the USENIX 2000 Annual Technical Conference*, San Diego, June 2000. http://www-sor.inria.fr/publi/ PFNMP_usenix2000.html.

[PST+97] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *16th ACM Symposium on Operating Systems Principles*, Saint Malo, France, October 1997.

[Sun] Sun Java. Java Message Queue. http://java.sun.com.

Contents

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399