Replication and Imprecision at work: Epsilon divergence control for wide-area cooperation

Georges Brun-Cottan and Liuba Shrira

April 2, 2000

Abstract

We present two new pessimistic divergence control protocols for cooperative replication, a replication model where each process submit its operations to a given one replica in charge of *synchronising* and *broadcasting* them to remote peers. In these protocols, replicas schedule their operations asynchronously regarding their peer, up to a given point, determined by the use of a metrics associated to each operation, at which they start to cooperate with each other.

We designed and implemented in Objective Caml [CM98] on top of Ensemble [Hay98] these two metrics based replication protocols. Each of them do use the metrics in a different way – which make them suited for very different kind of application: one was used for wide-area cooperation over a bounded resource. It has been proved correct in a companion paper. The second one was used for a multiuser distributed game over the Internet. It ensures a relaxed but bounded wide-area frame coupling between players.

The rationale of these two protocols is the same: they hide long-haul network latency to applications accessing the shared information by limiting the scheduling by each replicas of their non stable operations.

The heart of the two protocols is a stability tracker, based on a peer-to-peer metrics-based windows mechanisms. This stability tracker constrains the scheduling of locally issued operations.

We discuss the expected benefits of the two protocols. We present how the expected benefit of the imprecise bounded resource control protocol compares to a two months series of experiments made during the summer 1998 between USA, France and Israel.

1 Introduction

Wide-area cooperation faces a major problem: network latency. Cooperation involves interaction through a shared state. A recurrent problem is how to keep a good local interaction with the shared state while ensuring consistency on this state as required by the concurrent actions issued by the others. The first need demands for fast local interaction while the other demands for kinds of distributed agreement over a slow network.

This issue is currently addressed in various way, depending on the tradeoffs between local interactivity requirements, correctness and the availability to undo and redo actions. Some applications don't trade consistency, e.g. groupware (CSCW), others don't trade interactivity, e.g. distributed interactive simulations (DIS). Regarding wide area cooperation, the first category focus on optimistic strategies, the second one on smart best effort strategies (dead-reckoning, postponing of actions).

On a system point of view, the interactivity and the consistency requirement of a cooperative application resumes to the fast execution of locally initiated operations (interactivity), and the consistent distributed ordering of all operations (consistency).

On a wide-area system, a fast local execution demands for the presence of a close enough replica of the shared state. A consistent distributed ordering of operations requires for some distributed component that do constrain the execution of all operations.

This two requirements leads to the following cooperative replication. Every process accessing the shared information instantiate a replica of the state in its context[1]. Associated to each replica is a local scheduler. Client processes issue operations by submitting them to their local scheduler. This scheduler, does execute them on their local replica as well as on other replica. The set of local schedulers forms a distributed scheduler that do integrate the management of replication and concurrency.

Although cooperative replication have the potential to fulfill the two interactivity and consistency requirements, it does not address by itself their combined issue. How can operations be locally executed as fast as local operations and still some global ordering constraint be enforced?

The divergence control protocol is the key component that address this issue in a pessimistic way. Its rationale stems from the following observation: on a wide-area setting, strong consistency management is slow because such a management requires a perfect knowledge of the possible concurrent operations, even *before* deciding how to schedule them. That is, a *precise* distributed scheduler has to know about what to schedule before scheduling.

Our strategy is then straigthforward. Let's local schedulers take scheduling decisions before knowing about possible concurrent operations. The whole point is, of course, to restrain the local scheduling decision on the basis of dis-

[1] We may devise some intermediate scheme where one replica serves a local area. This can be achieved in a rather traditional client/server way.

tributed consistency constraints without knowing about the possible concurrent operations issued and scheduled by remote schedulers. This is our contribution.

Epsilon divergence control is build onto the three following hypothesis:

- The replicas are deterministics, that is, the initial replica state and the sequence of operations determine the state of a replica. Conversely, two replicas applying the same sequence of operations from a same state are guaranteed to be the same.
- There exists a sound quantitative divergence indicator, on which one can map the consistency requirement of the shared information.
- This indicator is independant of the state on which an operation is applied, that is, depends only on mutator operations by themselves.

We detail and justify these hypothesis further in the article, but we can spell out some intuitive examples: the divergence incurred by *inc* and *dec* operations of a counter can be the amount of increment or decrement done. The divergence incurred by *move* operations of an avatar in a DIS can be how much is the move, e.g. the cartesian distance. The point is that many kinds of cooperation does involve the sharing of quantitative information which allow for sound and intuitive numeric divergence indicators. The two applications developed further in this paper are such representatives.

The net effect of divergence control is to fasten the local execution of operation. This may benefits to applications in three different ways: i) it increases the throughput of applications whose speed was limited by consistency algorithmic. ii) It allows to run new applications that need consistency but until now was two slow to be used. iii) It brings some consistency to already fast application that did resign on consistency.[2]

Epsilon divergence protocol is a simple generic principle. It is build up from three local correctness criterions. We proved it correct in a companion paper. We implemented it as an OCaml module. It is small, around 300 lines of Ocaml code. We envision it as a basic component of the next generations of distributed systems.

We conducted a series of experiments during the 1998 summer, involving members located in USA, France and Israel sharing precise and imprecise N-bounded resources. We present a comparative analysis of these experiments along the expected analytical behavior of the protocol.

Controlling imprecisely concurrency and replication using a metrics is not a new idea. In particular, we have been a lot influenced by the long effort done by Calton Pu with ϵ-serialisibility on extending the transaction schedulers to make them "imprecision aware" [PL91, RP95, WYP97]. We merely took their work out of their original transactional setting to apply it to wide-area cooperation. The issue and its problematics turns out to be quite different. Epsilon serialisibility focuses on how to accept more transactional schedules

[2]The two first points looks very much the same but in the way they do face the additional problem of network bandwith limitation.

than do accept serialisibility. We do focus on how to get rid of knowing about the distributed concurrent operations and how this loose of information impacts the correctness of shared abstract data types used for cooperation.

2 Rationale

Although the rationale is simple – let's schedule with imprecise knowledge of concurrent operations – their is still the issue of bounding the imprecision without requiring to contact others replicas. The idea lies in the following observation: if, at the time it decides to schedule a locally initiated operation, a local scheduler can compute the divergence its operation will incur regarding the other peers, it can locally compute how much it can go forward without trespassing the application bound. If every replica plays this game, a global bound on the divergence is achieved.

There is several issues in trying to do this. First, of course, the application correctness must map a quantitative divergence criteria. Second, the divergence created by local operations regarding other peer's state must be locally computable. Third, this locally computed divergence must somehow relates to the global divergence bound the application is asking for. Let's precise these three issues.

Consistency as a divergence criterion Through intuition, by looking at when human faces imprecise knowledge and how he deals with it, by crafting and understanding two simple examples, an imprecise N-semaphore and the skeleton of a multi-user game, we envision there exists a significant number of cooperative applications that may translate some of their correctness in divergence requirements. It is not needed that the whole consistency criterion of an application be mapped in a divergence criterion. It is enough that the so translated part be the consistency criterion challenging the application responsiveness. We give answers on how divergence criterion can be managed together with the other consistency criterion in the specific case of our two examples. Although far to be definitive, we think these arguments are compelling enough.

Locally computable divergence Locally computing divergence implicitely relies on the two following hypothesis: first, the outcome of one replica being controlled by how replicas schedules diverge, replicas should behave deterministically. Second, given one operation, the divergence computed locally should be independant of the local replica state.

The first hypothesis is not a big constraint. Determinism is required by cooperative replication, and it is not clear how cooperation can be achieved over non deterministic behavior. The second hypothesis requires more explanations. The promise of the local computation is that, given a operation, it computes accurately how a remote peer that *does not know* about this operation would diverge from this same peer should it would *know* about this operation. The simplest way to cope with this requirement is for the divergence computation to depend only on the operation itself. As this operation is the same whichever replica executes it, the divergence associated to this operation will be the same whichever replica do compute this divergence. This requirement bring together an additional benefits and an important limitation.

Regarding benefits, this requirement allows to sum up the divergence contribution from more than one peer. This happen to be the basis to model imprecise agreement involving more than two members – a simple DIS application of this property is to bound how much members of a team may diverge on the observation of the position of a moving target.

Regarding limitations, this requirement, for instance, prohibits assignement operations to be controlled by our protocol. This kind of operation should typically be ordered with additionnal constraints, for instance totally ordered. This is usually a required contraint as assignement operations generally do not commute, and commutativity is required if we want the replicas to converge on the same state while allowing non identical ordering of operations.

Local criterions provides a global criterion How the individual bounds enforced by each replicas does relates to the bound on the global divergence? Is this global bound depend from the number of members? Badly enough, we do not have a general answer for these two questions, although we do have some hints. We can merely observe that the global bound depends on the number of cooperating members when more than two process agreement is needed. It is independant when the only form of agreement needed by the application is peer-wise. We give specific example of this when describing our test applications. The case where the global bound is dependant of the number of members raises an additional issue. The individual bound allocated must be able to change dynamically as new members join or quit.

We address these three issues as follows:

1. We first ask the application to map its consistency requirement by using a metrics. This embodies the two intertwined and not straightforward tasks: finding a sound metrics, that is expressing quantitatively the divergence incure by one operation, and then defining an accurate bound.

2. We then provide a component, the local scheduler that first keep track of an upper-bound of the divergence that the locally issued operations incur on the other replicas, and then blocks the local process progression when this computed upper-bound reach the bound locally assigned to the component by the application.

This strategy does not prevent per se the application to stop. The idea is first to evaluate on the average how much divergence does generates the operations making their way on the network. If the application can not afford to schedule local operations without their knowledge, there is no benefit. If the application *may afford* to schedule them, the benefits is two folds: first, on the average, the application goes as fast as best effort strategies. Second, it is still

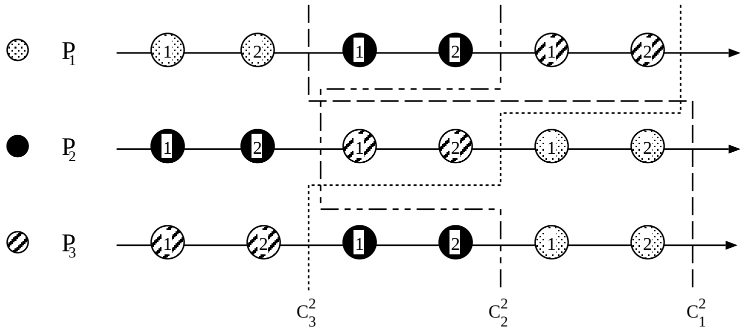

Figure 1: Bounded cuts. Each processes initiated two operations. There is six bounded cut on this execution. C_1^2, C_2^2 and C_3^2 are the second bounded cuts of respectively the first, the second and the third processes.

guaranteed a consistency bound. This is important in the frequent occasion where latency may increase over its average. In this case, the protocol will stop the application.[3]

In other words, epsilon divergence control allows applications to trade in a sound way, their consistency with their responsiveness.

3 Epsilon Divergence Control

Cooperative replication and determinism leads naturally to using replicated histories as a way of representing the evolution of a set of replicas. This representation abstracts the interprocess communication (IPC) implementation, which prove to be useful to grasp the consistency issues related to concurrency and replication. Roughly, an execution resumes to a set of histories, one for each replica. Histories from a same execution does shows the very same set of operations although, of course, not in the same order.[4]

Stating the divergence of histories, on such an execution model, happens to be not as straigthforward as stating about their equivalence – as asked, e.g. by active replication [PBS+88]. As Internet does not provide for deterministic time properties, we want to model the system as asynchronous. So, how do we state on the divergence of two histories?[5] How do we quantify this divergence? How do we control this divergence?

[3]The protocol may provide feedback on how far from the threshold is the local replica. This knowledge can be used to gently slow down the local replica before stopping.

[4]Each replica may be represented as a state machine, its history being its input sequence. Cooperative replication does achieve the *agreement* property [Sch90], but with no a priori on the *order* property.

[5]Checking for predicate over states represented by causal cuts are the traditionnal way to deal with this issues. But it inherently relies on the IPC model. We detail these issues in the companion paper.

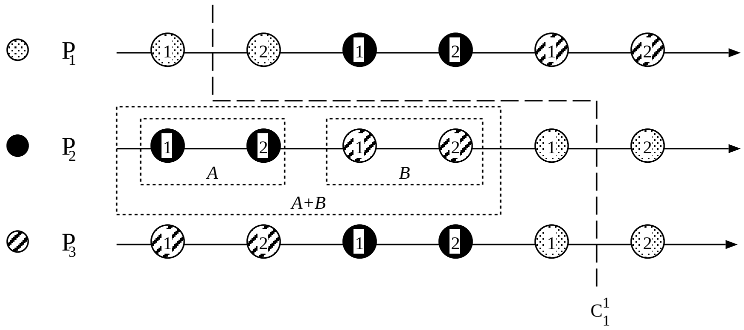

Figure 2: Divergence study between the process p_1 and the process p_2 on the specific bounded cut C_1^1, that is at the "time" where p_1 has initiated (and so executed) its first operation. A is the set of operation initiated by p_2 but not executed by p_1 at this "time". B is the set of operation initiated by p_3 executed by p_2 but not by p_1 at this same "time".

Stating on the divergence We propose to check for divergence on global points of the distributed execution defined by the execution of one given replicated operation by every replica. That is, points when one process *initiated* one operation and the other process *executed* it. We call those points, *bounded cuts.* Figure 1 illustrate these cuts. *Initiation* refers to the execution of one operation on the very process running the local scheduler to which this operation has been issued. So, initiation of an operation encompasses its execution *on its initiation process.*

Bounded cuts allow the following intuitive question to be asked: at the moment I did *initiate* an operation, what operations from the others I was not aware of? A key point is that bounded cuts allow this question, important regarding the notion of divergence, to be asked with no reference to time.

When comparing operations executed by two peers on a bounded cut, it turns out that two kinds of operations are of interest (see set A anf B, figure 2): the set of operations *initiated* by one peer and not executed by the other one (A) and the set of operations *executed* by one peers but not by the other one ($A+B$). The essential difference in power between the two sets as that the first one, A, does not model how much p_1 and p_2 disagree or diverge on the p_3 operations at that point. Controlling the divergence on the set A alone can not control more that peer-wise agreements. We will exhibit concrete application of this remark in our two examples application. From now, we focus on the general case where divergence must be controlled on the set $A+B$.

For a N processes execution and a given bounded cut defined by an operation initiated by a process p_i, divergence is stated by a N-tuple set of operations that we call the cut's execution delta. Each element of this tuple is a set equivalent to the set $A + B$, one for each peer of p_i.[6]. Similarly, we can define a cut's initiation delta, which is a tuple N-tuple whose each element is a set equivalent to the set A.

It may be worth to notice that bounded cuts capture distributed agreement. If we characterize agreement as the total ordering property, that is every operations are executed in the same order on every processes, agreement turns out to be *equivalent* to the case where the execution delta of every bounded cut of the execution is a tuple of empty set.

Quantifying on the divergence Once identified the operation accountable for the divergence, we want to quantify this divergence. Epsilon divergence control does quantify divergence by requiring first, for each operation to be assigned a metrics μ, and second, that the divergence incurred by a set of operation to be the sum of the indivual metrics of each operation in this set.[7] This last property is important. It allows each process to compute locally an upper bound of its divergence with its peers.

Let's consider the example illustrated by the figure 2. The set of operations A represents the contribution of the process p_2 in the divergence with the process p_1. The set of operations B represents the contribution of process p_3. The execution delta of this bounded cut is the tuple $(\emptyset, A \cup B, B \cup A)$. Let's assume that a metrics of one is assigned to each operation. The divergence between p_1 and p_2 on this bounded cut is $\mu(A \cup B) = 4$, as this is also the case between p_1 and p_3.

The ultimate goal being to define an upperbound on the divergence, we define the divergence associated on a given bounded cut to be the maximum of the metrics of each of the set in the execution delta of this cut. And in a similar logic, we define the maximum divergence of an execution to be the maximum of the divergence amongst all the bounded cut of this execution. On our example, the divergence would be 4, what we also called a 4-bounded execution ($\epsilon = 4$).

Controlling the divergence The key feature of the epsilon divergence control protocol is to rely on three *local* constraints.[8]

Let's first assume the simple following approach: each process that does initiate an operation keep tracks of its acknowledgement by each of its peer. After one process execute a remote operation, it sends back an acknowledgement to the source processor, that is the process that did initiate this operation.

So, each process, at any point c of its execution does maintain, for each of its peer a metrics, let's call it t^c, being the sum of the metrics of all its initiated operations not acknowledged by this peer. Let's call $t^c_{\max}$ the maximum of this metrics over all its peers. Assuming a system of N processes and a required

[6] We include the pair $p_i - p_i$ for simplicity. Its A set resumes trivially to the empty set.

[7] This proposition refers loosely to the notion of σ-additivity, although our "metrics" are not required to map on $\overline{\mathbf{R}}^+$. For instance, an upper lattice is fine.

[8] Their formalisation and their proof is available in a companion paper.

bound of ϵ such than $\tau = \epsilon/(N-1)$ and $\tau > 0$, we can informally state three local constraints:

1. A process p_i may *initiate* an operation at a given point c of its execution, only if $t_{\max}^{i,c} \leq \tau$.
2. On a process p_i, the processing of the acknowledgement by a process p_j of an operation o_i, must *precede* the execution on p_i of every operation initiated by p_j after it executed o_i.
3. A process may execute an operation only if it has executed its causal predecessors.[9]

The two first constraints are the core of the protocol. They limit the initiation of operations and so their sending to the remote peers. They bound how many operations a process may initiate without any agreement with its peers. The last constraint, a causal constraint, is needed to control the impact of third's members operation, e.g. for more than peer-wise agreement. It controls the remote execution order of operations. When together enforced, these constraints ensure that the execution is bounded by $(N-1)\tau$.

There is an issue in such a bound. It depends from the number of cooperating members. This means the protocol scale poorly. This means also that adding new members requires to recompute dynamically the individual threshold of each member, should the application bound ϵ be the same.

An interesting point is that however, there is a case where the bound became independant from the number of members ($\epsilon = \tau$). This is when the only operations of concern regarding divergence are the ones in the initiation delta (see figure 2, set A). This is of course an alternative divergence definition, but it may still be useful in primitive forms of coordination when two processes don't need to agree on third's one operations – e.g. in a DIS, two members of a team may disagree on whether a third player will be successfully shoot or not. Such an alternative divergence definition requires only the two first local constraints.

It should be clear, than our two first constraints are the one that do impact the application responsiveness. They may translate in process stalls. Let's precise a little bit more this issue. Responsiveness is usually tied to two properties: the abilities to *issue* operations at a desired rate and to fast *observe* the result of issued operations. Epsilon divergence control addresses the first one by allowing operations to be issued asynchronously with respect to remote replicas. It addresses the second in executing issued operation locally at once.[10] In other words, epsilon divergence control makes use of asynchrony, but in a controlled way rigorously defined by the application internal consistency constraints.

Last, two important remarks must be done. First, as these local constraints does assume $\tau > 0$, they can not be used to deal with the limit case $\tau = 0$. Practically, in our experiments, we use a sequencer based protocol to handle that case. Second, the divergence as controlled by this three constraints *might*

[9] An operation causally precedes an operation o_j, if it was executed before o_j initiation.
[10] If there is a stall, it is at issuing time.

nothing have to do per se with the intuitive divergence of the state of the replicas. These will become obvious on the next section, where some mutators operations (`Inc`) of the bounded resource are assigned a null metrics.

4 A Bounded Resource

We propose to share a N-bounded resource over a wide-area setting. Such an abstraction is *per se* useful for distributed applications. Integrated in a control structure such a semaphore, it mays, e.g. control the number of floating licenses in a wide-area virtual enterprise or limit the number of participant in virtual classroom, virtual races or games. Alone, it forms a basis for implementing the cooperation of applications sharing any form of quantitative information. As an example, the N-bounded resource combines three interesting aspects:

- it has a well-understood semantics,
- it exhibit a straightforward metrics, and,
- it has provision for two didactic, though useful, relaxations on its semantics.

A bounded resource is an abstract data type associated with one state `S` and two operations, `Inc` and `TestAndDec`. Informally, its serial specification would look like the following:

Resource:	*initialisation*	$\mathtt{S} \leftarrow N$
	`Inc`	$\mathtt{S} \leftarrow \mathtt{S} + 1$
	`TestAndDec`	if $\mathtt{S} > 0$ then $\mathtt{S} \leftarrow \mathtt{S} - 1$

Let us distribute such a bounded resource using cooperative replication. We present three different semantics of this replicated version. For each of them we outline how they translate in term of the divergence control and present their intuitive impact on the application responsiveness.

Atomicity Atomicity gives the replicated version of the bounded resource the same semantics than the centralized one. Not only, the resource state is guaranteed to be positive, $S \geq 0$, but as `Inc` and `TestAndDec` operations are done in a single total order, each of them is guaranteed to be executed on a state that do reflect all previous distributed operations.

Regarding the distributed implementation, atomicity translates into the total ordering of all operations, `Inc` as well as `TestAndDec`. Such a constraint maps into a zero-bounded execution, assuming all operations are assigned a metrics of one.[11]

The throughput of such implementation is tied to the network. On a sequencer based implementation, the average operation latency is the average round-trip time to the sequencer. With such a semantics, the use of cooperative replication model itself is arguable.

[11] This has not practical impact on the implementation, as zero-bounded execution are not addressed by our protocol. Our implementation uses a sequencer to deal with that case.

TestAndDec Atomicity This first relaxation stems from the following observation: the essence of a bounded resource is in its invariant $S \geq 0$. Postponing the execution of `Inc` operations does not impact this invariant. Although the state S of each replica may not accuratly represent the global bounded resource state[12] it still represents a lower bound of it.

Regarding the distributed implementation, this TestAndDec atomicity translates into the total ordering of `TestAndDec` operations but only require the causal ordering of `Inc` with respect to previously initiated `TestAndDec` operations.

Analysing the throughput of such implementation is a little bit more complex. Latency of `TestAndDec` operations is the same as for the atomic implementation. Latency of `Inc` operations is better. First `Inc` operations are initiated at local processor speed – they don't have to wait for any communication. Second the remote execution itself of a `Inc` operation is likely to be faster as causal ordering imply less communication and delivery delay than total ordering. This semantics is a kind of win-win relaxation.

ϵ-atomicity In this last relaxation epsilon divergence protocol is used to control the order of `TestAndDec` operations. The idea is to allow the invariant to be $S \geq -\epsilon$ without changing the implementation of the `TestAndDec` operation that is using the same $S > 0$ guard. The state S of each replica represent then a lower bound of the global state *accurate* at ϵ. Of course, this last relaxation does change the bounded resource semantics as it allows the case $0 > S \geq -\epsilon$.

Regarding the distributed implementation, ϵ-atomicity translate into the epsilon divergence control of `TestAndDec` operations and the causal ordering of `Inc` with respect to previously initiated `TestAndDec` operations.

We only sketch the throughput benefits of this implementation, as two full sections of this article are devoted to it. `Inc` operations performs the same way as in the first relaxation. Assuming a metrics of one is assigned to `TestAndDec` operations, in the ideal condition where ϵ account for all operations not known as being stable,[13] `TestAndDec` operation may proceed as fast as `Inc` operation.

Epsilon divergence protocol provides two interesting behaviors: first it has no latency impact[14] when it is not needed, that is roughly when there is no more than *epsilon* concurrents `TestAndDec` operations. Second, it synchronizes processes only for the amount of time not cover by ϵ, that is for the amount of concurrent operations above ϵ.

Last, we may notice that `TestAndDec` operations backwards commute with others `Inc` and `TestAndDec` operations[15]. That is, a posteriori, the order of `Inc` and `TestAndDec` operations does not matter regarding the eventual convergence of replicas – it is achieved by the mere operations semantics. In the bounded

[12] Informally, the global state is the difference between the sum of all *initiated* `Inc` operations and the sum of all *initiated* `TestAndDec` operations. It is sound only on some points of the execution which include bounded cuts.

[13] We say an operation is stable if it has been *executed* on every process of the system.

[14] In sparse bandwith setting, the processing of acknowledgement may become an issue.

[15] We refer to WEIHL definitions of forward and backward commutativity [Wei88]

resource case, epsilon divergence controls the *observations* of remotely issued `TestAndDec` operations as returned by the "read and test part" of a locally initiated `TestAndDec` operation.[16]

5 License Manager

Using an epsilon bounded resource to represent the number of free license available is based on the following rational. We envision that future system will be build out from numerous third party component dynamically exchangeable. In such a system, each component or set of component provided by a given IT provider can be charge on the basis of its load or concurrent usage. The system can be developped world-wide as is already the management system of organisation whose offices spread out several continent. In such an environement, the ability for component to be dynamically purchased, replaced and updated will go along with a custom charge management that will allow company to pay accordingly to the usage they of the service. We call this accounting service a license manager. The idea is for this license manager to grant access to the spread out components accordingly to what have been payed by the organisation. The idea is that the usage profile should be accurate enough to map the charging model but yet have no impact on granted access.

Such a licence manager should be able to dynamically replicate itself, ideally one replica on each side of a long-haul communication network. It should also be able to respond with little latence to requests. Our epsilon-bounded resource is a good candidate for this service. Our protocol allows low-latency operation and has provision for a dynamic renegociation of the threshold among the group of replica.

We sketch out two different model of license management: in the first one, the dynamically loaded service came out with the bound and accuracy (ϵ) embedded, one for all, at the time the component is uploaded. Although the bound and ϵ are static, the threshold may evoluate depending upon the number and usage of replicas existing at a given moment in the system. In the second one, we envision a kind of lease model, where the service provider (Electronic newspaper, JIT compiler, Active caches) keeps tracking the usage of its service by keeping a replica. The client can then be charge monthly depending on how much it make simultaneous use of the service. Epsilon control allows limit the impact of the monitoring on the service performance but still provide accurate enough to charge the client. Furthermore, the provider can easily change the accuracy as well as the bound itself according e.g. to contract updates.

In a world where digital information service can be duplicate and/or accessed concurrently easily with no almost no infrastructure cost for the client, the ability to have a performant accounting mechanism is of primary importance. It allows a service to be charged in an fair way both for the client and the service provider/developer.

[16] A `TestAndDec` operation is an atomic agregation of read, test and decrement operations.

6 Protocol

6.1 Principles

From a principle point of view, the protocol works like an asynchronous pipeline, the size of the pipeline being defined by the threshold τ. Application consumes pipeline space by emitting new operations. The divergence control protocol produces pipeline space each time an operation in the pipeline is known as being stable on every peers. While the pipeline is not full, operations execute at local speed – they are locally scheduled with no delay. When the pipeline is full, the application blocks until some space become available.

The amount of time the application blocks is the service time of the divergence control protocol. For a given node, it is driven by several factors, including the network latency between the node and its farther peer and the rate at which the application emits operations on this node. The operations in the pipeline are operations not known as being stable, that is operations that have been executed locally but whose execution at remote peers has not been acknowledged.

In a nutshell, our divergence protocol controls the amount of asynchrony available to the application. The operation are asynchronous only regarding their execution on remote peers. When the pipeline is not full, the application may execute synchronously and immediately all its operation on its local node. The size of the pipeline defines the amount of divergence allowed between nodes.

The idea is to make the size of the pipeline comparable to the $\mathrm{RTT}_{\max}$. The network latency behave so like a capacity, and the asynchrony is then used to hide the network latency.

The effect of hiding latency can then be used in two different ways by the application: it can be used to render operation execution fast – roughly at local speed. It can also be used to allow new applications, that need higher operation rate, to run on a wide area setting. By decreasing the execution speed of an operation, the protocol support applications that require operation rates unreachable,e .g. with a sequencer. Of course, a main side effect of this usage is that by doing so, an application also increase its load on the network, and may do so, up to a point that do congest the network.

6.2 Algorithm

The protocol is two part. The first part manages the metrics-based windows. It controls the sending of locally issued operation to remote schedulers as well as their local scheduling. The second part handles causality. It controls the scheduling of remotely issued operations.

The causal part is a subset of the traditional causal order. It enforces the only two following order relationship : `TestAndDec` < `TestAndDec` and `TestAndDec` < `Inc`. `Inc` are not ordered with each other and `TestAndDec` operation are not ordered with respect to `Inc`[17]. This is done in a straightforward way: `Inc`

[17]The current implementation do FIFO ordered these operations but for different reason

```
let pushOp op =
  for i = 0 to (N - 1) do
     windows[i] <- windows[i] + op.mu;
  done

let ackOp ack i =
  windows[i] <- windows[i] - ack.mu

let maxOfWindows () =
  let maxThreshold := 0 in
  if (windows[i] > maxThreshold) then maxThreshold := t;
  return maxThreshold
```

Figure 3: The three windows manipulation primitives used by the metrics part of the protocol.

operations does not increment the local vectorial clock of the scheduler. All operations are executed with respect to the order so defined.

The metrics-based part is built over a simple stability tracker of `TestAndDec` operations: each scheduler keep tracks of `TestAndDec` operation locally issued. Each scheduler acknowledged the execution of `TestAndDec` to the scheduler that issued it. The epsilon control is done the following way: before scheduling one locally issued `TestAndDec` operation, a scheduler compute for each of its peer the cumulative metrics of the operation still not acknowledged by this peer. The maximum of these cumulative metrics over all its peer is its current threshold usage. The scheduler may schedule a local operation if its current threshold usage once added up to operation's metric is still below its threshold τ.

The application issues operations to the protocol through the function `issue`. The protocol submit operation to the application through the function `execute`. `execute` returns true when the operation has been applied on the local replica of the bounded resource. `Inc` operations can always be applied. `TestAndDec` operations can only be applied if the replica state is strictly positive ($\mathtt{S} > 0$). Operation not executed by the application are kept on the scheduler queue and resubmit later on to the application when the protocol state allows it.

The protocol uses two communication primitives: `fbcast` and `send`. `fbcast` is a FIFO multicast. It does not deliver messages back to the sender. `send` is point to point. It delivers messages in FIFO order with respect to both `send` and `fbcast` messages. The protocol provides the two corresponding delivery primitives, `deliverSend` and `deliverFbcast` called by the transport protocol upon reception of a message.

(reliable delivery and acknowledgement constraint).

```
let deliverSend ack sender =
    queues[sender].enqueue ack;
    processState()

let deliverFbcast op sender =
    queues[sender].enqueue op;
    processState()

let issue op =
   if localSchedule op then
      processState();
   else
      pendingOperation := op
```

Figure 4: The three entry point of the protocol.

```
let localSchedule op =
   if (op.mu + maxOfWindows () <= τ) and (execute op) then
   begin
      pushOp op;
      op.ts := siteTime;
      if op.mu > 0 then
         siteTime[myid] := siteTime[myid] + 1;
      fbcast op;
      return true;
  end else
  return false;

let processState () =
   do
     scheduled := false;
     for i = 0 to (N - 1) do
       if queue[i] not empty then
          scheduled = processOneQueue i;
     done;
     if some pendingOperation then
        scheduled = scheduled or (localSchedule pendingOperation);
   while (scheduled = true)
```

Figure 5: The core sender part of the protocol.

Elements of the protocol For the sake of the simplicity, we assume a fixed number N of scheduler[18]. Each scheduler is assigned by the application a thresh-

[18] Dynamic membership slightly complexify the clock management. It requires furthermore some interaction with the group membership service to identify view change. This is take in

```
let processOneOperation id op
  if op is an ack then
    ackOp op op.sender;
    return true;
  else if GE(siteTime, op.ts) then
     if execute op then
     begin
        if op.mu > 0 then
        begin
           siteTime[op.sender] := siteTime[op.sender] + 1;
           ack.sender := myid;
           send op.sender ack
        end;
        return true;
     end else
        return false

let processOneQueue i =
  stateChanged := false;
  do
     op := queue[i].top;
     if (processOneOperation i op) then
     begin
       stateChanged := true;
       queue[i].pop ();
       op := queue[i].top;
     end else
       break; (* exit the loop *)
  while (queue[i] not empty)
  return stateChanged
```

Figure 6: The core delivery part of the protocol.

old τ.

Causal ordering uses vectorial clocks. Each scheduler maintain its local clock `siteTime`. The element i of the local clock is incremented each time a `TestAndDec` operation issued by the scheduler i is executed by this scheduler. Each scheduler stamps the operations by its local clock before sending them to its peer. The predicate `GE(X,Y)` is true if and only if $\forall i X_i >= Y_i$.

Each operation is assigned three elements: `origin` identifies the process which did initiate this operation. `ts` is the timestamp of this operation as assigned by the scheduler to which this operation has been issued. `mu` is the

our implementation, but none of these are related to the metrics management principle.

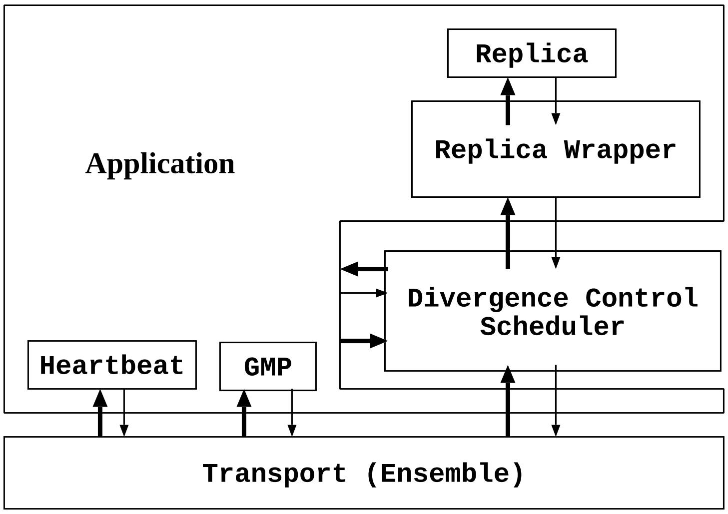

Figure 7: The divergence control in situation. Bold arrows identifies the Ensemble upcall paths. There is no multithreading. The whole application is driven by Ensemble events (transport management, group management, heartbeat).

---

metrics associated to this operation. It is assigned by the application.

Stability management is reduced to what is needed for the purpose of the metrics management. Each time a scheduler execute a remote operation o issued to the scheduler i, it send back to this scheduler i and acknowledgment `ack` stamped with the metrics `o.mu` (`ack.mu` = `o.mu`). An important point is that acknowledgement can not be processed at once but must be enqueued. This is needed to ensure the required end-to-end FIFO property between operations issued by one scheduler and its acknowledgment.

Metrics management relies on a vector of metrics `windows`, one component for each scheduler. Upon the execution of a locally issued operation o, a scheduler i increments all the component of its metrics `windows` by `op.mu`.

In addition, each scheduler maintains a set `queues` of N FIFO queues, one for each scheduler of the system. The queue i of the scheduler i is used to enqueue operations locally issued. Other queues hold operations received from the remote schedulers.

Figures 3 and 4 define the basic primitives. Figures 5 and 5 constitute the core of the scheduler.

7 Implementation

We have implemented two protocols: one totally order `TestAndDec` operations while the other one implements epsilon divergence control. These two protocols are implemented as Objective Camel [CM98] modules on top of Ensemble [Hay98], a group communication platform developed at Cornell. A benchmark application has also been developped in Objective Caml. It implements a shared semaphore replicated according to the cooperative replication model. According to some parameter the application, choose at run time which divergence control protocol to use.

Although the two protocols are independant, each of them have specific requirement on the transport protocol as well as on the transport interface itself. For instance, the total ordering of `TestAndDec` operation, required modification of the Ensemble transport interface, so that `Inc` operations does not suffer the cost of total ordering. The espilon divergence control requires also FIFO ordering between point-to-point Acknowledgement messages and the FIFO multicast messages used to broadcast operations. Those additional requirement have been easily implemented as additional Ensemble layers.

The figure 7 pictures the architecture of the application. The flow control is a bit unusual as all the application activity occurs in Ensemble upcalls. Although this is not a requirement, it makes the application far simpler and faster, at the expense of a small startup overhead.

8 Performance

One important characteristics of our divergence control algorithms require a special care: it is partially asynchronous, and so the service time of a given operation depends on the state of previous operations issued at this node. Although these operations have already been locally executed, the divergence control protocol keeps track of their execution on all remote peers. The time a node may postpone the execution of a locally issued operation, what we called the operation service time, depends on the local knowledge that do have this node regarding the execution of the predecessors of this operations on its remote peers.

Also, the raison d'etre of the protocol, being in the end-to-end control of the application level, we design our bench application to execute its operations synchronously: operations are *locally* issued and executed sequentially. More precisely, the application iterates infinitly on the two following steps:

- A *think* step while the application does not issue any operation. It only executes remote operations asynchronously received from other nodes that has been scheduled by its local scheduler.
- An *execution* step while the application initiates an operation, passing it down to the scheduler, waiting for its scheduling and executing it *locally*.

The think time was a parameter of the application, while the execution time is, of course, determined by performance of the coordination protocol. The number of replica, the kind and the number of operation they execute was also parameter of the application. The benchmark starts when the replica quorum is reached.

8.1 Expectation

There is two expected benefits from the protocol:

- the network latency is hidden, and
- the application throughput is increased.

The network latency is hidden because operation issued while the protocol threshold is below the authorized threshold $\tau_{\max}$, are executed locally with no delay. For a given member, The application throughput is increased because it depends no longer from the round trip time ($\mathrm{RTT}_{\mathrm{seq}}$) between this member and the sequencer but from the product $\tau\mathrm{RTT}_{\max}$, $\mathrm{RTT}_{\max}$ being the round trip time between this member and its farther peer.

Let assume the following hypothesis:

- a local and remote execution times el and er,
- a constant think time T,
- a constant threshold τ,

For a given operation k the service time is :

$$\mathrm{St}_k = \mathrm{el}_k + \max(0, \mathrm{RTT}_{k-\tau}^{\max} + \mathrm{er}_k - \tau\mathrm{T} - \sum_{i=1}^{\tau-1} \mathrm{St}_{k-i}) \tag{1}$$

If we assume a negligible local execution time (el = er = 0), under the ergodic hypothesize, we expect:

$$E(\mathrm{St}) = \max(0, E(\mathrm{RTT}^{\max}) - (\tau - 1)E(\mathrm{Stp}) - \tau\mathrm{T}) \tag{2}$$

with $E(\mathrm{Stp})$ being the expectation of the service time of the $\tau-1$ predecessors of an operation. Furthermore if $E(\mathrm{Stp}) = E(\mathrm{St})$, the service time get even simpler:

$$E(\mathrm{St}) = \max(0, [E(\mathrm{RTT}^{\max}) - \tau\mathrm{T}]/\tau) \tag{3}$$

However, this last case is unlikely to occur. (1) implies a stwo step behavior and wide-area setting implies a high variance of $\mathrm{RTT}^{\max}$. In such a setting, even in the favorable case $E(\mathrm{RTT}^{\max}) < \tau\mathrm{T}$, we expect some operations to stall. The behavior of these stalls is expected to be given by (2), St and Stp being restricted to those stalling operations.

8.2 Experiments

We ran a two monthes serie of experiment between USA, France and Israel incolving our implementation of the bounded resource in two flavors: one using a total ordering of `TestAndDec`, the other using our epsilon divergence control

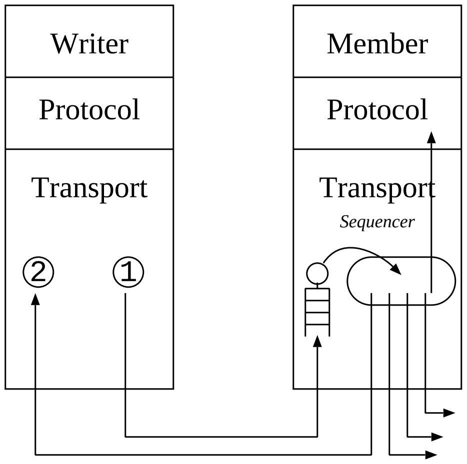

Figure 8: RTT in the sequencer experiments. RTT is the time between the *abcast* request to the local total layer, event 1, and the local delivery of this abcast, event 2 – that is $t_2 - t_1$. Every writer, but the sequencer itself, records their RTT with the sequencer.

---

protocol. The experiment was live, done during day and night with normal network and machine load.

All operations issues in these experiments were `TestAndDec` operations. Each writer issue 5000 of this operations. The metrics associated with `TestAndDec` operation was one. The results presented was collected with a setting of three members and two concurrent writers. Regarding the reference experiment totally ordering operations, the sequencer node does not issue any operations. It was located at the best averaged position, providing a latency of roughly half the latency diameter of the settings.

We made some experiments to evaluate the evolution the protocol load on the latency. These results shows a high variance rendering difficult the interpretation. It seems nevertheless to indicate that the acknowledgement does incur a significant load on the network.

8.2.1 Collected Informations

All the experiments logged the following set of information:

- The total elapsed time for issuing, initiating and executing each of its own operation[19] as well as the total time for executing all the operations issued in the system.
- The total CPU consumption, system and user modes.

[19] Only writers do gather and log these operations specific informations.

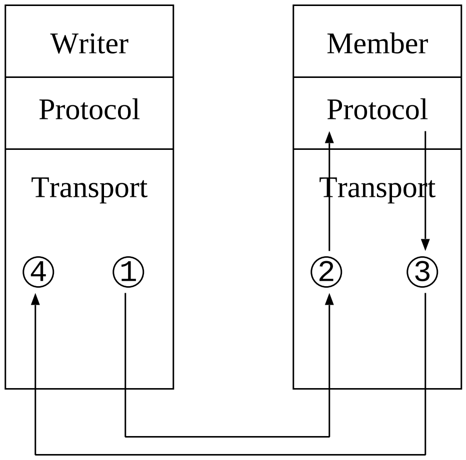

Figure 9: RTT in the divergence control experiments. RTT is the time $t_4 - t_1 - (t_3 - t_2)$. Event 1 is the *fbcast* request to the local multicast layer. Event 2 is the corresponding delivery of this multicast to member. Event 3 is the send of the divergence control acknowledgement back to the writer. Event 4 is the delivery of this acknowledgement to the writer. Every writer RTT records its RTT with all of the members.

- The RTT latency.

The RTT latency is interpolated and requires more explanations. Regarding the totally ordered protocol, the RTT is the elapsed time between the send of the abcast request to the sequencer and the delivery of the abcast on the same issuing node (see figure 8). In the totally ordered protocol, the RTT is given only between the nodes and the sequencer.

Regarding the divergence control, the RTT uses the basic property of the divergence control protocol that each operation is send with a fifocast and acknowledged by a point to point acknowledgement. The RTT is the elapsed time between the send of the fifocast and the delivery of the corresponding acknowledgement less the local time on the remote member between the delivery of the fifocast and the sending of the acknowledgement (see figure 9). In this protocol, the RTT is computed only at writers processes but is given for each peers of writers-members. The two protocol of the RTT measure when used in similar environment setting shows reasonably similar value. As for other measures, the high standard deviation make conclusion difficult: A very same experiment, even long (more than one hour), running at a different time of the day can show significantly different RTT.

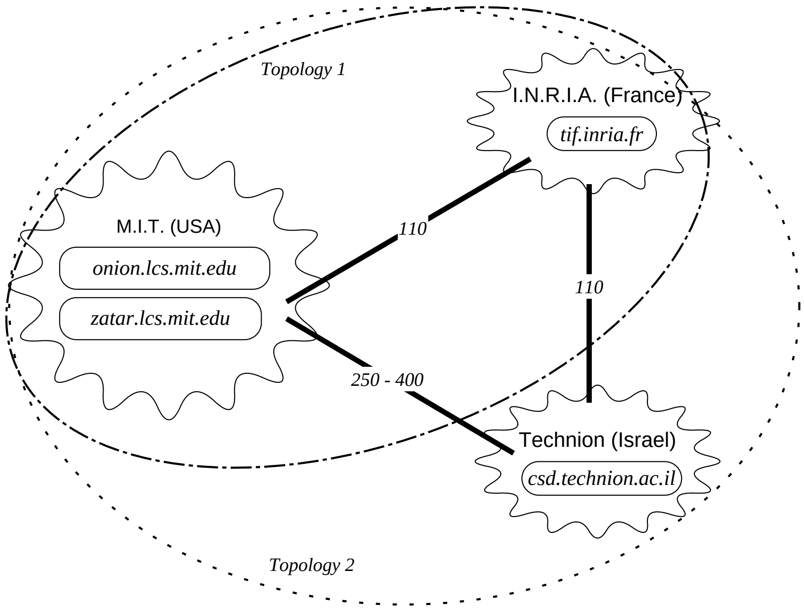

Figure 10: The two topologies used in our experiments. Links are stamped with a gross average of their RTT over the time of the experiment.

8.2.2 Experimental Setting

We run our experiments on two different communication topology invoving three continents (see figure 10): topology 1 involves USA and France. Topology 2 invloves USA, France and Israel.

8.2.3 The Communication Links

The two topologies studied involve three long-haul communication links[20]. The table 1 presents the typical behavior of these links as measured during the experiments.

Independantly of the link involved, low averages are associated to low standard deviations.

Independantly of the link involved, the experiments running the higher application rates (between 50 and 200 op/sec) and higher threshold (between 40 and 160) consistently show high RTT average and high standard deviation. On

[20] By link, we do not mean a static well defined IP routing path, but rather the set of routes used by our nodes during the experiments.

Link	Round Trip Time (RTT) in milliseconds	
	Mean	Standard deviation
M.I.T. –I.N.R.I.A.	100 – 160	20 – 160
M.I.T. – Technion	200 – 600	50 – 200
I.N.R.I.A. – Technion	100 – 120	20 – 30

Table 1: Typical RTT measured during the experiments.

the two links involved with USA, the average RTT as well as the standard deviation goes above 1 sec[21]. The third link between France and Israel still shows a similar behavior although far less spectacular.

Further investigations done on special purpose independant experiments between USA (Brandeis university) and France, show that losses are responsible for this dramatic increase of the latency: between USA and France, all RTT above 300 msec, was due to message loss. The increase in latency is due to the time to retransmit and deliver the lost messages.

The two protocols use two distinct fifo reliable protocols provided by Ensemble. The Ensemble `Mnak` protocol is used to ensure fifo reliable multicast while the Ensemble `Pt2pt` protocol is used to ensure fifo reliable point to point communications. The `Mnak` protocol is driven by explicit retransmission requests by the receiver while the `Pt2pt` protocol is driven by periodic retransmission of not acknowledged messages by the sender. We found the setting of this timeout retransmission parameter(`pt2pt_sweep`) to change dramatically the impact of the message lost on the latency.

Although increasing the retransmission rate at first glance seems to address the problem, it does have an important side effect on the bandwith consumption. In our experiments, decreasing the retransmission period time from 5 sec to 500 msec increased by four the number of retransmission, resulting in that up to 10% of the messages was retransmitted – for an effective loss rate of 1.4%[22].

While the observed impact of the reliable protocol on the average latency may be not surprising, it is a strong reminding that current reliable protocols seems to be more tuned for bandwith requirements than for latency requirements. As latency tend to be more and more the important issue, current reliable protocols might have to be either parametrized differently or redesigned.

8.2.4 The Nodes

Latency results are inherently very sensitive to scheduling. Two of our test machines (tif.inria.fr and csd.technion.ac.il) are service machines. The table 2

[21] Those experiments have been done from Brandeis university (U.S.A.)

[22] Losses are heavily asymetrics. There was of 0.1% in the USA → France way and 1.3% the other way around. Although the asymmetric behavior of the trans-atlantic link is well-known, the published results tend to show a reverse asymmetry: USA → France used to be worse than the other way. We don't have a good explanation for this behavior.

Nodes	induced	faced
Tif†	induced	faced
Lachesis†	induced	faced
Atropos	induced	faced
Csd†	induced	faced

Table 2: Two indicators of the machine load during the experiments. Nodes with a dag are writers. The left column shows the load induced by the protocol itself. The right column shows the load faced by the experiments due to other not experiment related processes.

presents two measures of the load of these machines.

The first measure is the elapsed CPU time used by the experiments. It shows the load that the protocol imposes on the bench and to the machine running our experiment.

The second one shows the elapsed real time due to the processing of an operation and its acknowledgement by a remote member. As this path is short – few function calls and no system calls – it is not expected to take more than a few microseconds. Any significant delay is likely to be due to a descheduling. We shows a normalized value of the average of these delay. This value shows indirectly the load not due by our experiment but by other processes running on the same machine than our experiments.

8.3 Results

9 Related Work

10 Conclusion

References

[CM98] Guy Couzineau and Michel Mauny. *The Functional Approach to Programming.* Number ISBN 0-521-57183-9. Cambridge University Press, 1998. `http://pauillac.inria.fr/caml/`.

[Hay98] Mark Hayden. *The Ensemble System.* PhD thesis, Cornell University, jan 1998. `http://www.cs.cornell.edu/Info/projects/Ensemble/`.

[PBS+88] D. Powell, G. Bonn, D. Seaton, P. Verissimo, and F. Waeselynck. The delta-4 approach to dependability in open distributed computing systems. In *International Symposium on Fault-Tolerant Computing (FTCS'88)*, pages 246–251, Washington, D.C., USA, June 1988. IEEE Computer Society Press.

[PL91] Calton Pu and Avraham Leff. Replica control in distributed systems: An asynchronous approach. In *Proceedings of the 1991 International conference on the Management of Data*, ACM SIGMOD Records, pages 377–386, Denver, CO, USA, May 1991. ACM SIGMOD, ACM Press.

[RP95] Krithi Ramamritham and Calton Pu. A formal characterization of epsilon serializability. *IEEE Transactions on Knowledge and Data Engineering*, 7(6):997–1007, December 1995.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. Technical Report TR 86-800, Cornell, 1986/1990.

[Wei88] William E. Weihl. Commutativity-based concurrency control for abstract data type. *ACM Transactions on Computer Systems*, 37(12):1488–1505, December 1988.

[WYP97] Kun-Lung Wu, Philip S. Yu, and Calton Pu. Divergence control algorithms for epsilon serializability. *IEEE Transactions on Knowledge and Data Engineering*, 9(2):262–274, March/April 1997.