High-Performance Transaction Processing in Sprint

Lásaro Camargos[⋆,†] Fernando Pedone[†] Marcin Wieloch[†]

[⋆]State University of Campinas (Unicamp), Brazil

[†]University of Lugano (USI), Switzerland

University of Lugano
Faculty of Informatics
Technical Report No. 2007/01
January 2007

Abstract

Sprint is a middleware infrastructure for high performance and high availability data management. It extends the functionality of a standalone in-memory database (IMDB) server to a cluster of commodity shared-nothing servers. Applications accessing an IMDB are typically limited by the memory capacity of the machine running the IMDB. Sprint partitions and replicates the database into segments and stores them in several data servers. Applications are then limited by the aggregated memory of the machines in the cluster. Transaction synchronization and commitment rely on total-order multicast. Differently from previous approaches, Sprint does not require accurate failure detection to ensure strong consistency, allowing fast reaction to failures. Experiments conducted on a cluster with 32 data servers using TPC-C and a micro-benchmark showed that Sprint can provide very good performance and scalability.

1 Introduction

High performance and high availability data management systems have traditionally relied on specialized hardware, proprietary software, or both. Even though powerful hardware infrastructures, built out of commodity components, have become affordable in recent years, software remains an obstacle to open, highly efficient, and fault-tolerant databases.

In this paper we describe Sprint, a data management middleware targeting modern multi-tier environments, typical of web applications. Sprint orchestrates commodity in-memory databases running in a cluster of shared-nothing servers. Besides presenting Sprint's architecture, we discuss three issues we faced while designing it: (a) how to handle distributed queries, (b) how to execute distributed transactions while ensuring strong consistency (i.e., serializability), and (c) how to perform distributed recovery.

Distributed query processing in Sprint differs from that in traditional parallel database architectures in two aspects: First, high-level client queries are not translated into lower-level internal requests; instead, as a middleware solution, external SQL queries are decomposed into internal SQL queries, according to the way the database is locally fragmented and replicated. In some sense this approach resembles that of multidatabase systems, except that the component databases in Sprint have the same interface as the external interface, presented to the clients. Second, Sprint was designed for multi-tier architectures in which transactions are pre-defined and parameterized before the execution. This has simplified distributed query decomposition and merging.

Sprint distinguishes between *physical servers*, part of the hardware infrastructure, and *logical servers*, the software component of the system. These come in three flavors: *edge servers (ES)*, *data servers (DS)*, and *durability servers (XS)*. A physical server can host any number of logical servers. For example, a single DS only, both a DS and an XS, or two different instances of a DS. *Edge servers* receive client queries and execute them against the data servers. *Data servers* run a local in-memory database (IMDB) and execute transactions without accessing the disk. *Durability servers* ensure transaction persistency and handle recovery.

IMDBs provide high throughput and low response time by avoiding disk I/O. IMDBs have traditionally been used by specific classes of applications (e.g., telecommunication). Current performance requirements and cheaper semiconductor memories, however, have pushed them to more general contexts (e.g., web servers [19], trading systems [31], content caches [13]). In most cases, applications are limited by the memory capacity of the server running the IMDB. Sprint partitions and replicates the database into segments and stores them in the recovery of several data servers. Applications are limited by the aggregated memory capacity of the servers in the cluster.

Data servers in Sprint never write to disk. Instead, at commit time update transactions propagate their commit votes (and possibly updates) to the transaction participants using a persistent total-order multicast protocol implemented by the durability servers. As a consequence, all disk writes are performed by durability servers only, ensuring that during normal execution periods disk access is strictly sequential. Durability servers are replicated for high availability. If a durability server crashes, data servers do not have to wait for its recovery to commit update transactions; normal operation continues as long as a majority of durability servers is operational. Data servers are replicated for performance and availability. When a data server crashes, another instance of it is started on an operational physical server. The new instance is ready to process transactions after it has fetched the committed database state from

a durability server.

Sprint distinguishes between two types of transactions: *local* transactions access data stored on a single data server only; *global* transactions access data on multiple servers. Local transactions are preferred not only because they reduce the communication between servers and simplify pre- and post-processing (e.g., no need to merge the results from two data servers), but also because they do not cause distributed deadlocks. Although in theory there are many ways to deal with distributed deadlocks, in practice they are usually solved using timeouts [11]. Choosing the right timeouts for distributed deadlock detection, however, is difficult and application specific. We solve the problem by ordering global transactions and thus avoiding distributed deadlocks.

Finally, Sprint does not rely on perfect failure detection to handle server crashes. Tolerating unreliable failure detection means that the system remains consistent even if an operational server is mistakenly suspected to have crashed, another one is created to replace it, and both simultaneously exist for a certain time. Since Sprint ensures serializability even in such cases, failure detection can be very aggressive, allowing prompt reaction to failures, even if at the expense of false suspicions.

We have implemented a prototype of Sprint and conducted experiments using MySQL 5 in "in-memory" mode (i.e., no synchronous disk access) as the local data engine at data servers. Experiments with TPC-C revealed that:

- When the database was large enough to fill the main memory of 5 DSs, Sprint outperformed a standalone server by simultaneously increasing the throughput and reducing the response time by 6x.
- In a configuration with 32 DSs, Sprint processed 5.3x more transactions per second than a standalone server while running a database 30x bigger than the one fitting in the main memory of the single server. If the database on the single server doubles in size, growing beyond its main memory capacity, then Sprint can process 11x as many transactions per second.
- The abort rates due to our distributed deadlock prevention mechanism for configurations ranging from 1 to 32 DSs were quite low, aborting fewer than 2% of TPC-C transactions.
- Terminating transactions using a total order multicast proved to be 2.5x more efficient than using an atomic commit protocol with similar reliability guarantees.

Experiments with a micro-benchmark allowed us to evaluate Sprint's performance under a variety of workloads. Results demonstrated that:

- When the database was large enough to fill the main memory of 8 DSs, Sprint had better throughput than a standalone server for all possible combinations of global/local and read-only/update transactions. In some cases, the improvement in throughput was more than 18x.
- When the database was small enough to fit in the main memory of a standalone server, Sprint provided better throughput than the single server in workloads dominated by local transactions, both in cases in which 50% of the transactions updated the database and when all transactions only read the database.

- Experiments revealed that abort rates are highly dependent on the percentage of global transactions in the workload, up to 25% of aborts when all transactions are global, and less sensitive to the operations in the transactions. Workloads with 50% of global transactions were subject to 13% of aborts.

The rest of the paper is structured as follows. Section 2 states assumptions and presents definitions used in the paper. Section 3 overviews Sprint's architecture. Section 4 details the transaction execution and termination protocol. Section 5 discusses recovery. Section 6 describes our system prototype. Section 7 presents experimental performance results. Section 8 reviews related work, and Section 9 concludes the paper. Proofs of correctness are presented in the Appendix.

2 Background

2.1 Servers, communication and failures

Sprint runs in a cluster of shared-nothing servers. Physical servers communicate by message passing only (i.e., there is no shared memory). Logical servers can use both point-to-point and total order multicast communication.

Total order multicast is defined by the multicast(g, m) and deliver(m) primitives, where g is a set of destinations and m is a message. It ensures that (a) if a server delivers message m, all operational destination servers will also deliver m *(agreement)*, and (b) if two servers deliver messages m and m' they do so in the same order *(total order)* [12].

Physical servers can fail by crashing but do not behave maliciously (i.e., no Byzantine failures). A server may recover after a failure but loses all information stored in main memory before the crash. Each server has access to a local stable storage (i.e., disk) whose contents survive crashes. The failure of a physical server implies the failure of all the logical servers it hosts.

We do not make assumptions about the time it takes for operations to be executed and messages to be delivered. The system employs *unreliable failure detection* [4]: (a) failed servers are eventually detected by operational servers, but (b) an operational server may be mistakenly suspected to have failed (e.g., if it is too slow).

2.2 Transactions and databases

A *transaction* is a sequence of SQL statements terminating with a commit or an abort statement. Each transaction has a unique identifier. A transaction is called *read-only* if it does not modify the database state, and *update* otherwise.

Sprint guarantees the traditional ACID properties [11]: either all transaction's changes to the state happen or none happen *(atomicity)*; a transaction is a correct transformation of the database state *(consistency)*; every concurrent execution is equivalent to a serial execution of the same transactions using a single copy of the database *(isolation* or *one-copy serializability)*; and the effects of a transaction survive database crashes *(durability)*.

Transactions are scheduled at individual IMDBs according to the *two-phase locking* rule [3].

3 Sprint architecture

Figure 1 overviews the architecture of Sprint. Clients submit transactional requests to edge servers. Requests regarding the same transaction should be submitted to the same ES; new transactions can be started on different ESs. Edge servers are started up and shut down according to load-balancing and fault-tolerance requirements.

Database tables are partitioned over the DSs. An assignment of the database to DSs and the mapping of DSs onto physical servers is called a *database configuration*. Data items can be replicated on multiple DSs to allow parallel execution of read operations. This comes at the expense of write operations, which should modify all replicas of a data item. The database configuration changes when a DS fails and a new instance of it is created on a different server.

The *Query Decomposer* receives SQL statements from the clients and breaks them up into simpler sub-statements, according to the current *Database Configuration*. The *Dispatcher* interacts with the data servers, using either point-to-point or multicast communication, and ensures proper synchronization of transactions. The *Result Assembler* merges the results received from individual data servers and returns the response to the client.

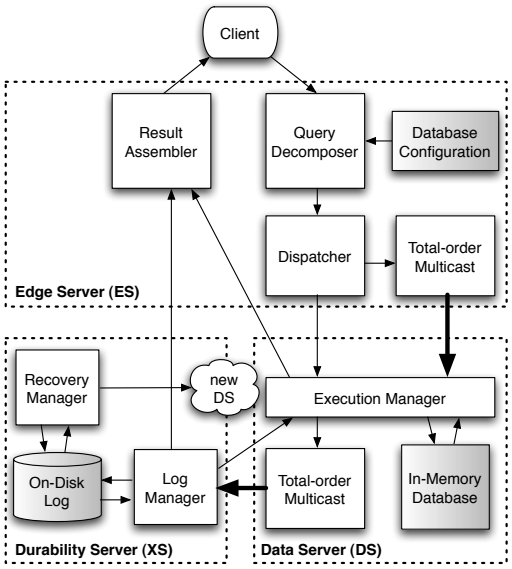

Figure 1: Sprint architecture

The *Execution Manager* receives SQL statements, submits them to the local *In-Memory Database*, collects the results, and returns them to the corresponding edge server. The Execution Manager also participates in the transaction termination protocol: Update transactions are multicast to the durability servers upon termination.

All permanent state is stored by the durability servers. This includes both the database state and the database configuration. The *Log Manager* implements stable storage with a sequential *On-Disk Log*. Since all disk writes are performed by the durability servers only, during most of the execution disk access is strictly sequential. The Log Manager informs the edge servers and the Execution Manager about the state of terminating update transactions. Rebuilding the state of a failed data server from the log is performed by the *Recovery Manager*.

4 Data management in Sprint

In the absence of failures and failure suspicions, transaction execution in Sprint is very simple. For clarity, Sections 4.1 and 4.2 explain how the protocol works in such cases. Section 4.3 discusses how failures and failure suspicions are handled.

4.1 Transaction execution

Edge servers keep two data structures, *servers* and *status*, for each transaction they execute. The first one keeps track of the data servers accessed by the transaction and the second one stores the current type of the transaction: *local* or *global*. These data structures exist only during the execution of the transaction and are garbage collected once it is committed or aborted.

The execution of local transactions is straightforward: every SQL statement received from the client is forwarded to the corresponding DS for processing and the results are returned to the client. If a transaction executes an operation mapped onto more than one DS or onto a DS different than the one accessed by a previous operation, it becomes global. When it becomes global, the transaction is multicast to all DSs—in fact only the transaction id is multicast. Each global transaction is multicast only once, when the ES finds out that it is global; subsequent requests are sent to the DSs using point-to-point communication (see Figure 2).

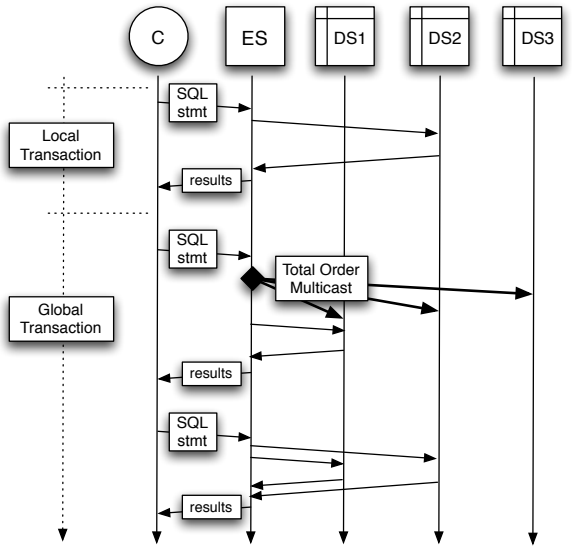

Figure 2: Transaction execution

The multicast primitive induces a total order on global transactions, used to synchronize their execution and avoid distributed deadlocks. When a global transaction T is delivered by a DS_k, the server assigns it a unique monotonically increasing sequential number $seq_k(T)$. The Execution Manager at DS_k implements the following invariant. Hereafter we say that two global transactions *conflict* if they access data on the same DS and at least one of the transactions updates the data; and that a DS *receives a transaction* T when it first receives an operation for T.

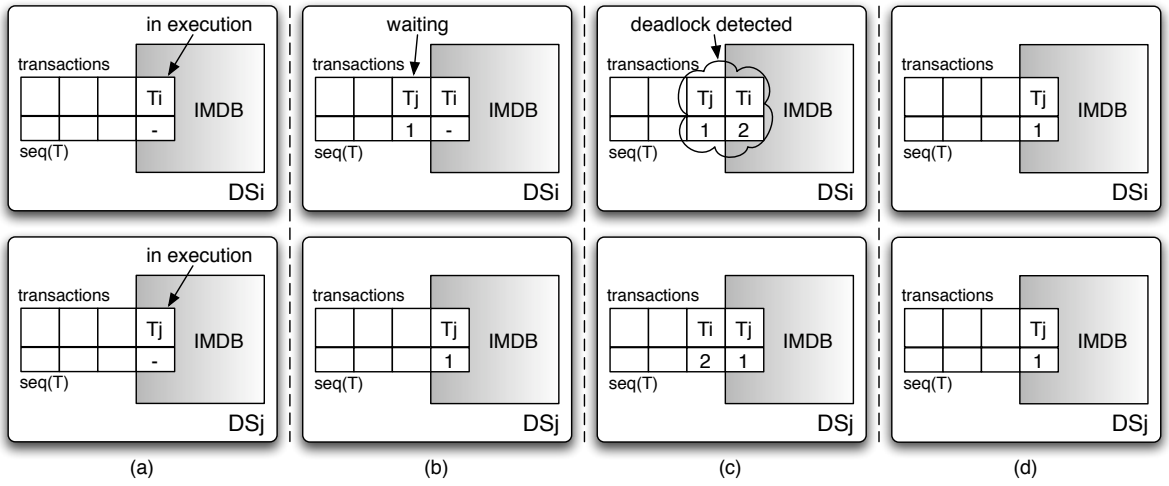

Figure 3: Deadlock resolution

- *Execution order invariant.* Let T_i and T_j be global conflicting transactions. If T_i is received after T_j by DS_k, then $seq_k(T_i) > seq_k(T_j)$.

Sequential numbers define the executing order of global transactions. Intuitively, serializability is guaranteed by the local scheduler at each DS and the fact that no two DSs order the same transactions differently—we present a formal correctness argument in the Appendix. Local deadlocks are resolved by the in-memory database executing the transactions. Distributed deadlocks are resolved by avoiding cycles in the scheduling of transactions.

Consider for example the execution in Figure 3. In step (a) T_i and T_j execute locally on DS_i and DS_j, respectively. In step (b), T_j requests a data item on DS_i, is multicast to DS_i and DS_j, and receives sequential number 1 at both DSs. T_j becomes global now. In step (c), T_i requests access to DS_j, is multicast, and assigned sequential number 2 at both DSs. To keep the algorithm's invariant, T_i is aborted. This happens first on DS_i, which notifies the edge server handling T_i so that other DSs involved in the transaction abort it too. In step (d) T_j executes on both DSs.

The price to pay for simple and timeout-free deadlock resolution is the abort of transactions that may not in fact be deadlocked. In the example in Figure 3, assume that T_i's second request accesses a data item on DS_l, $l \neq j$ and $j \neq i$. There is no deadlock involving T_i and T_j now. However, to enforce the execution order invariant, DS_i will still abort T_i. In our experiments we observed that despite this simplified mechanism, abort rates were low (see Section 7).

4.2 Transaction termination

Read-only transactions are committed with a message from the ES to the DSs involved in the transaction. The transaction terminates when the ES receives an acknowledgment from each DS.[1] If a DS fails and cannot send the acknowledgment, the ES will suspect it and abort the

[1]To see why acknowledgments are needed, assume that a transaction finishes after a message is sent from the ES to the DSs concerned. Let T_i be a read-only transaction, T_j an update transaction, and X and Y data items

transaction. Acknowledgments are needed to ensure correctness despite DS failures. They are discussed in more detail in Section 5.

Termination of update transactions is more complex. Committing an update transaction involves the XSs to guarantee that the committed state will survive server failures. Termination of update transactions is based on total order multicast. Figure 4 illustrates termination in the absence of failures and suspicions. The ES handling the transaction sends a prepare message to all DSs involved in the transaction (message 1). Each DS multicasts its vote to the ES, the participating DSs, and the XSs (messages 2a and 2b).

Although any total order multicast algorithm can be used in Sprint, message 2b in Figure 4 zooms in on the execution of the Paxos protocol, used in our prototype. Both messages exchanged among servers and the disk accesses done by the XSs, which play the role of "acceptors" in Paxos parlance, are depicted. Figure 4 includes two optimizations to the Paxos algorithm, as described in [17]: (a) XS_1 acts as the "coordinator" and has previously executed "Phase 1" of Paxos, and (b) the replies from the acceptors are directly sent to all "learners" (i.e., multicast destinations).

If a DS is willing to commit the transaction, it multicasts together with its vote the update statements executed by the transaction. Upon delivering a message from every participating DS or an abort message, each destination server can determine the outcome of the transaction: commit if all DSs have voted to commit it, and abort otherwise. If the outcome is commit, each DS locally commits the transaction against its IMDB.

Terminating update transactions with a total order multicast serves two purposes: First, it provides a simple way for data servers to render their state persistent (on the XSs). Second, it avoids the shortcomings of more traditional atomic commitment protocols (e.g., 2PC may block in the presence of failures [3]; 3PC is quite complex and cannot tolerate false failure suspicions [9]). Finally, notice that although conceptually each DS should execute a multicast with its vote, in practice the coordinator (XS_1 in Figure 4) can batch the votes of several DSs and send them all together to the acceptors, reducing the number of messages and disk writes.

4.3 Termination under failure suspicions

Most of the complexity involved in terminating a transaction in case of suspicion of a participating DS stems from the possibility of wrong suspicions, that is, the ES suspects a DS that did not crash. Handling such cases is complex because the suspected DS could have multicast its vote to commit the transaction when the ES suspects it and acts accordingly (e.g., locally aborting the transaction and telling the client). As a result, some servers may take the DS's vote into account to determine the outcome of the transaction, while others may not, reaching an inconsistent decision.

Ensuring that all concerned servers reach the same outcome when terminating an update transaction is done as follows: If an ES suspects the failure of a DS during transaction termination, it multicasts an abort vote on behalf of the DS. As before, a transaction can only commit if all participating DSs vote to commit it. But now, the votes considered in the decision are the first ones delivered for each DS. For unsuspected DSs, only one vote will be delivered; for suspected DSs, there will be possibly multiple votes. In any case, the total order multicast

in DS_X and DS_Y. T_i reads X from DS_X, which then fails and is replaced by DS'_X. T_j then updates X and Y in DS'_X and DS_Y, and commits. Finally, T_i reads Y and commits. The execution is not serializable: T_i sees X before T_j and Y after T_j.

ensures that all destination servers deliver transaction votes in the same order, and therefore reach the same decision. For simplicity, hereafter we refer to the first vote delivered for a DS as simply *the DS's vote for the transaction*, irrespectively of its sender.

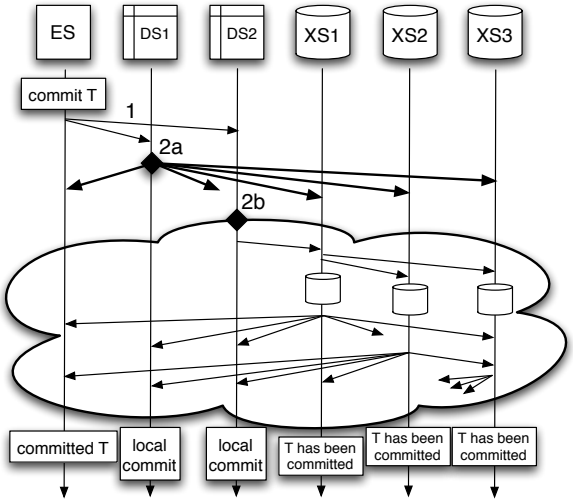

Figure 4: Terminating an update transaction

5 Recovering from failures

5.1 Edge servers

If an ES fails during the execution of a transaction, the DSs involved will eventually detect the failure and abort the transaction. If the ES fails during the termination protocol, the transaction may end up committed or aborted, depending on when the failure occurs: If the ES's request to terminate the transaction reaches all participating DSs, these are willing to commit the transaction, and their votes are delivered before any other votes for the transaction, then the outcome will be commit.

A new instance of an ES can be promptly created on any physical server. During bootstrap, the ES sends a message to one of the XSs asking for the current database configuration. The ES will be ready to process requests once it receives the database configuration.

5.2 Data servers

5.2.1 The basic idea

Sprint's approach to recover a failed DS is to simply create another instance of it on an operational physical server. Since DSs run "pure" IMDBs (i.e., configured to avoid disk accesses), there is no database image to be restored from the local disk after a crash. As a consequence, a new copy of the failed DS can be deployed on any physical server using the state stored by the durability servers.

While this strategy simplifies recovery, care is needed to avoid inconsistencies. For example, consider the following execution. Transaction T_i reads data item d_i from DS_i and transaction

T_j reads data item d_j from DS_j. Then, both DSs fail and are replaced by new copies, DS'_i and DS'_j. T_i requests to modify d_j, stored on DS'_j, and T_j requests to modify d_i, stored on DS'_i. Thus, both transactions become global and are multicast. Neither DS knows about the past reads, and so, T_i and T_j are not properly synchronized. Moreover, since DS'_i has not received any operation from T_i, T_j's sequence number is not checked against T_i's, and its write operation is locally executed. Similarly, DS'_j executes T_i's write operation. As a result, both T_i and T_j are committed, violating serializability.

5.2.2 Avoiding inconsistencies due to recovery

Sprint avoids inconsistencies by ensuring that *a transaction can only commit if the DSs it accesses are not replaced during its execution.* We achieve this property by using *incarnation numbers* and *incarnation number vectors* to compare the configuration of the system when a transaction starts and when it tries to commit. Incarnation numbers are unique identifiers of each instance of a DS; they can be implemented by simply counting how many times a DS has been replaced or "incarnated". An incarnation number vector contains one incarnation number per DS in the system.

When a transaction starts, it is assigned the vector with the most up to date incarnation numbers the ES has seen. At the termination time, the incarnation number of each DS involved in the transaction is compared against its vector to check whether the transaction can commit. The revisited condition for commit is as follows:

- *Consistent termination invariant.* A transaction T can commit if for each DS participating in T
  - DS's vote is to commit T, and
  - DS's incarnation number matches its entry in the transaction's incarnation number vector.

In our prototype the database configuration is extended with an incarnation number vector. When a transaction starts, the ES hosting it assigns its current view of the vector to the transaction and, later, the vector assigned to the transaction is sent along with the prepare message sent by the ES as part of the procedure to terminate the transaction (message 1 in Figure 4). Each DS can check locally if the conditions to commit the transactions are met.

When an ES suspects the failure of a DS, it multicasts a *change-DS* request to all servers together with the identity of the physical server to host the new instance of the DS. Upon delivering this request, all servers consistently increase the incarnation number of the particular DS and update the database configuration. Since vote messages, multicast by the DS itself or by the ES on behalf of the suspected DS, are ordered with respect to change-DS messages, all servers reach the same outcome regarding a transaction.

Acknowledgment messages sent by the DSs as part of the commit of a global read-only transaction return the current DS's incarnation number. To make sure that the transaction has observed a consistent view of the database, the incarnation numbers in the transaction's vector should be up to date. Acknowledgments allow edge servers to identify possibly inconsistent states.

Figure 5 depicts the termination of transaction T when one of the participating DSs, DS_1, fails. After the failure, ES eventually suspects DS_1 and multicasts the message *change-DS*$_1$.

Upon delivery of this message, the servers update the incarnation number of DS_1 and the database configuration. Since T's incarnation vector cannot match the new state of the system, both ES and DS_2 will decide to abort T. The physical server that will host the suspect DS also delivers message *change-DS*$_1$ and starts the recovery procedure, described next.

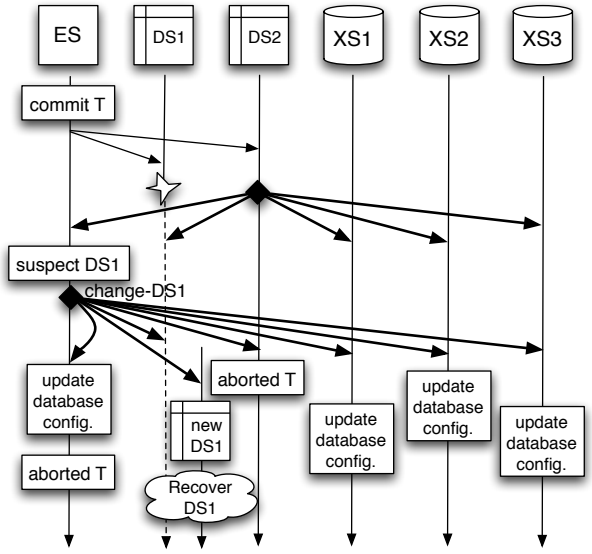

Figure 5: Recovery of a data server

Finally, notice that while sequential numbers are assigned to transactions to avoid distributed deadlocks, incarnation number vectors are assigned to transactions to ensure consistent execution in the presence of DS recovery.

5.2.3 Rebuilding the state of failed DSs

In principle, the state of a failed DS can be recovered from another DS if it is replicated. In any case, the database state of a DS can be recovered from the logs kept by the XSs. In this case, the new DS needs an initial database image and the missing updates to be applied to the image. After installing the initial database image, the log entries can be locally replayed.

Sequential numbers locally assigned to global transactions by the failed DS are not recovered by the new DS, which simply resets its sequence number counter. If a DS receives a request from a global transaction and it misses its sequential number because the transaction was delivered before the DS instance was created, the DS simply aborts the transaction.

Quickly recovering a failed DS is important for availability. Transactions requesting a data item stored on the failed DS cannot execute until the server is replaced. In our experiments, recovering a failed DS running the TPC-C benchmark can take a few minutes. If higher availability is required, then DSs should be replicated. Although read-only transactions accessing a replicated data item will succeed provided that at least one DS storing the item is operational, update transactions will fail since all replicas of the item must be available in order for the transaction to be committed.

To increase the availability of update transactions when replication is used, a failed DS should be removed from the database configuration. This is done by having the ES multicast a

request to increase the incarnation number of the suspected DS and exclude it from the current database configuration.

5.3 Durability servers

Recovering an XS is straightforward. Once the server is operational, it can take part in multicast instances—this follows directly from the Paxos protocol. Delivered messages missed by the recovering XS can be retrieved from operational XSs. Besides implementing Paxos, XSs also play a role in recovering failed DSs. In order to do it efficiently, each XS periodically creates an image on disk of the current database state. This state is built from the messages delivered by the XSs, as part of the termination protocol of update transactions.

6 Implementation

In the following we describe our system prototype. The middleware is implemented in Java with one independent multithreaded module per logical server. Point-to-point communication uses TCP and UDP sockets; multicast relies on a communication library built on top of UDP sockets and IP multicast.

6.1 Edge servers

Clients submit pre-defined parameterized SQL statements to ESs, which split them by looking up the Database Configuration directory. Each SQL statement type is decomposed into "sub-SQL statements" to be sent to the concerned DSs. The directory also contains a pointer to the post-processing procedure used to merge the results returned by the DSs in case of complex queries. Post-processing procedures should be written by the application programmer or could be chosen from a library of standard procedures.

The mapping of client statements onto data server statements depends on the type of the SQL request and the way the database is partitioned and replicated among the DSs. For example, the directory in Figure 6 assumes that DSs are numbered from 0 to $N-1$ and tables "employee" and "country" are hash partitioned on their primary keys ("employee.id" and "country.id" modulo N). Statement type 1 is a simple lookup executed only by the DS storing the required record. Statement type 2 is executed by all DSs. In both cases the ESs have no post-processing to do; the results are simply forwarded to the clients. Statement type 3 is a join between two tables. The original statement is split into two sub-statements. Procedure "f_join(...)" merges the results before returning them to the client. Both sub-statements are sent to all DSs. Tables relatively small, such as "country", could be stored entirely on one or more servers.

Optimizing the database partitioning and automatizing the breaking up of complex queries accordingly has been the theme of much research in parallel database systems (e.g., [16, 21, 35]). Some of these techniques could be integrated into the edge server logic, but this is out of the scope of the present work.

6.2 Data servers

Data servers receive and execute transactional requests. Transactions are received by two threads, one for point-to-point and the other for multicast communication, and enqueued for

Type	SQL request (input)	sub-SQL request (output)	Data server	Post-processing
1	SELECT * FROM employee WHERE id=?	SELECT * FROM employee WHERE id=?	id % N	-
2	UPDATE employee SET salary=salary*1.1 WHERE salary < ?	UPDATE employee SET salary=salary*1.1 WHERE salary < ?	0..(N-1)	-
3	SELECT * FROM employee, country WHERE employee.country = country.id AND employee.salary > ?	SELECT * FROM employee WHERE salary > ?	0..(N-1)	f_join(...)
		SELECT * FROM country	0..(N-1)	
4	COMMIT	COMMIT	all concerned	-
...	...	...	...	...

Figure 6: Database Configuration directory

execution. Threads from an execution pool take requests from the queue and submit them to the local database or to the XSs, in case of commit requests. As part of a commit request, ESs send to each DS involved in the transaction the identity of the participating DSs. This information is transmitted to XSs together with the updates performed by the transaction. DSs can then know when they have received the votes from all DSs involved in the transaction.

6.3 Durability servers

Durability servers participate in the total order multicast protocol and build images of the database stored by the DSs to speed up their recovery. The multicast library consists of a set of layers implementing Paxos [17]. The library is tailored for LANs, making heavy use of IP multicast. Total order multicast is implemented as a sequence of consensus executions. In each instance, XSs can decide on a sequence of multicast messages.

Durability servers build images of the database space using local on-disk databases. SQL statements of committed transactions are submitted to the local database asynchronously. Although building this image prevents the access to the disk of an XS from being strictly sequential, by keeping this procedure on a minority of them, the sequential pattern is still kept in the critical path of transactions. Ideally, the database image is directly extracted from the database via some special API call. When not possible, it can be extracted with a simple "SELECT *" query.

7 Performance evaluation

7.1 Experimental setup

In all experiments we used a cluster of nodes from the Emulab testbed [32]. Each node is equipped with a 64-bit Xeon 3GHz, 2 GB RAM (although we had slightly less than 1 GB available in our experiments), and a 146GB HDD. We performed experiments with up to 64 nodes, all connected through the same network switch. The system ran Linux Red Hat 9, and Java 1.5.

Data servers used MySQL in "in-memory mode", meaning that synchronous disk accesses were switched off during the execution and the database of a single DS fitted in its main memory.

Under these conditions, servers cannot recover from failures like on-disk databases do. Instead, when a DS recovers from a crash, any local state written to disk asynchronously by MySQL is discarded.

Durability servers used a local Berkeley DB Java Edition to store information on disk. Berkeley DB does not keep a database image on disk, but only a log. All access to disk is strictly sequential. During the recovery experiments, XSs also had MySQL to build the recovery of DSs images.

7.2 Multicast scalability

We initially evaluated the scalability of the total order multicast protocol. Table 1 reports the average and the maximum number of successfully multicast messages, i.e., delivered by all destinations, per second (mps). There are $2 * f + 1$ acceptors, which are also learners.

Receivers (learners)	Message Size	Resilience		
		$f = 1$	$f = 3$	$f = 5$
8	16 B	6476 (6851)	5190 (5446)	—
16	16 B	6051 (6651)	4042 (4875)	3298 (3623)
32	16 B	5537 (5897)	4390 (4218)	3924 (4034)
8	1000 B	1182 (1663)	766 (1027)	—
16	1000 B	1177 (1749)	801 (902)	434 (606)
32	1000 B	1170 (1588)	800 (943)	468 (595)
64	1000 B	1145 (1389)	791 (915)	447 (508)

Table 1: Performance of the multicast protocol

During the experiments, multiple threads constantly multicast messages to all receivers; there was no other traffic in the network. As part of the protocol, acceptors log messages on disk before delivering them (see Figure 4). In all cases, messages were delivered within a few milliseconds.

The performance of our total order multicast primitive is highly sensitive to the resilience, but not much to the number of receivers. There is not much variation with the number of receivers since our library is based on IP-multicast and all nodes are connected to the same physical switch. For the experiments in Sections 7.3 and 7.4 we used a multicast using 3 acceptors.

Smaller messages lead to higher throughput since more of them can be bundled into the payload of a single network message. We ran experiments with 16-byte and 1000-byte messages, corresponding respectively to the average size of the micro-benchmark and the TPC-C transactions we used in our experiments.

7.3 The TPC-C benchmark

For TPC-C, we range partitioned all but the Items table according to the warehouse id. The Items table is only read, and replicated on all data servers. With this partitioning, about 15% of the transactions were global, and 92% of them updated the database. Each DS stored locally 5 TPC-C warehouses. A warehouse in MySQL has around 100MB, resulting in a database of approximately 500 MB per DS. There were 6 to 8 ESs and 3 XSs, each one running on a dedicated physical server.

7.3.1 The impact of a cluster of IMDBs

Figure 7 illustrates the advantages of placing the database in the main memory of servers. It also shows compares Sprint to a standalone MySQL database. Sprint has constant response time for increasing database sizes, varying from 500 MB to 15 GB in these experiments. We have isolated the effects of parallelism by having a single client execute transactions. As a consequence, throughput = 1 / response time.

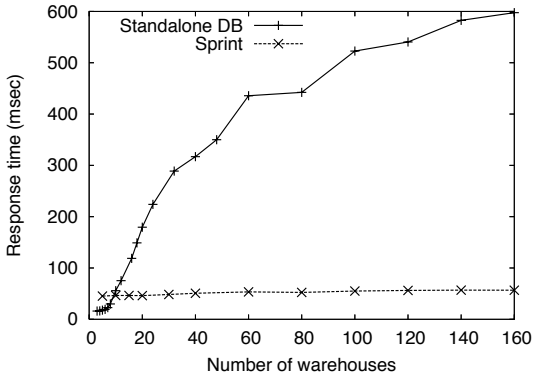

Figure 7: TPC-C with a single client

The response time of transactions executing in Sprint increases slightly with the number of DSs since there are more receivers in the multicast protocol. In our experiments, DSs vary from 1 to 32. A configuration with 1 DS has response time of about 45 msec, and a configuration with 32 DSs has response time of 56.7 msec, corresponding to throughputs of 22.2 tps and 17.6 tps, respectively. For 25 warehouses, for example, Sprint improves both the throughput and the response time of a standalone server by 6x.

When the database fits in the main memory of a standalone server, the overhead introduced by Sprint makes it unattractive. However, this rapidly changes as the database grows. If the standalone server had more main memory, the curves in Figure 7 would cross at some point right of the current crossing point, but the trend would be the same.

7.3.2 Throughput and response time

Figure 8 depicts the attained throughput in transactions per second (tps). The system load was varied by varying the number of clients. In most cases, 100 msec corresponds to the maximum throughput.

Besides presenting configurations of Sprint with a varying numbers of DSs, we also show the performance of the system when transactions are terminated with Paxos Commit, an atomic commit protocol [9] ("8AC" in the graph). Paxos Commit has the same latency as Two-Phase Commit but is non-blocking [9]. It allows data servers to contact durability servers directly (see Figure 4), saving one communication step when compared to total order multicast, where data servers should contact the coordinator first. We show the curve for 8 DSs only since this was the best performance achieved.

Although Paxos Commit saves one communication step, it introduces more messages in the network and does not allow the coordinator to bundle several application messages into a single

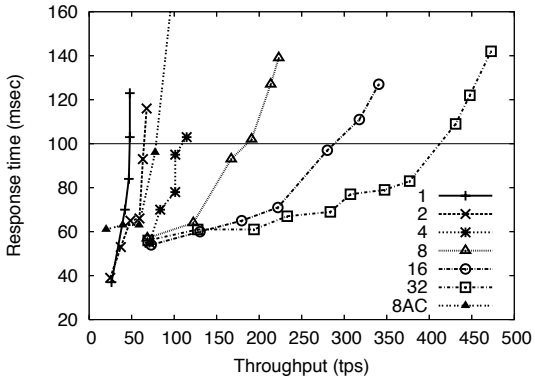

Figure 8: Throughput and response time of TPC-C

proposal. As a result, in Sprint it proved to be less efficient than terminating transactions with a total order multicast.

Table 2 compares Sprint with a standalone database when multiple clients submit transactions. The throughput is for response times of 100 msecs. We also include the performance of Sprint with Paxos Commit ("Sprint (AC)").

In a configuration with 32 DSs, Sprint processed 5.3x more transactions per second than a standalone server (i.e., 412 tps/77.5 tps) while running a database 30x bigger than the one that would fit in the main memory of the single server (i.e., 15.3GB/506MB). If the database on the single server doubles in size (i.e., Sprint's database is 15x bigger), then the throughput becomes 11x that of the single server.

	#WH	#DS	tps	DB size
Sprint	5	1	47	506 MB
	10	2	65	994 MB
	20	4	110	1.9 GB
	40	8	186	3.8 GB
	80	16	289	7.6 GB
	160	32	412	15.3 GB
Sprint (AC)	40	8	75	3.8 GB
Standalone Database	5	—	77.5	506 MB
	10	—	37.4	994 MB

Table 2: Sprint vs. standalone database (TPC-C)

7.3.3 Abort rates

We also evaluated the abort rates of TPC-C transactions. The values depicted in Figure 9 were collected during the same experiments as the ones shown in Figure 8.

The configuration for 1 DS represents the aborts induced by the local database, mainly due to local deadlocks and contention. Local aborts happen because in TPC-C many transactions read and modify a small table with one entry per warehouse. Since there are 5 warehouses per DS, with a single DS, the contention in this table is high. As the number of DS increases,

the probability of real deadlocks decreases. Our mechanism to prevent deadlocks aborts more transactions than needed but with TPC-C the overall number of aborts is quite low.

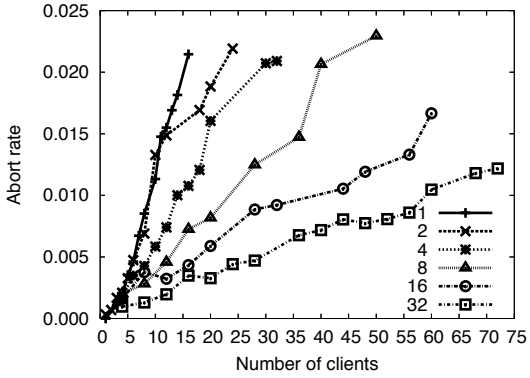

Figure 9: Abort rate of TPC-C transactions

7.3.4 Scalability

Figure 10 compares the throughput of Sprint to the throughput of a standalone MySQL as the database grows. The number of warehouses determines the size of the database, as depicted in Table 2. In all points in the graph we selected the number of clients that maximizes throughput. When the database fits in the main memory of a single server, then Sprint's overhead makes it unattractive. Configurations with 2 DSs or more can store a database bigger than the main memory of the single server. In such cases, Sprint has better performance than a standalone server. High throughput is also due to the parallel execution of transactions in multiple servers.

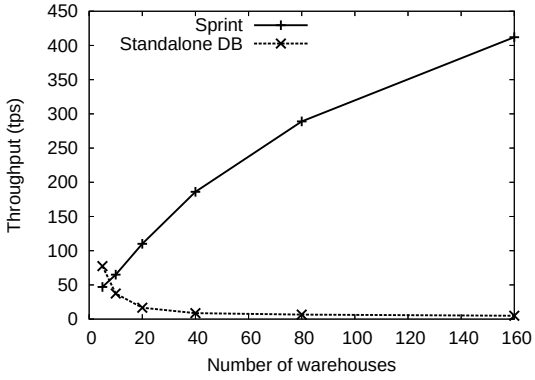

Figure 10: Scalability of Sprint and standalone DB

7.3.5 Recovering from failures

To evaluate the time needed by data servers to recover after a failure, we performed experiments with 28 TPC-C clients running for maximum throughput in order to generate large logs, the worst scenario for recovery. Clients issued transactions through 8 edge servers, and 8 data servers were online. In these experiments the database images were created every 10 minutes, what is roughly the time to increase the size of the logs by 54 MB (i.e., approximately 6.75 MB per DS). The recovery was started after 10 minutes of execution, right before the database images were created.

For this scenario the recovery takes, in average, 186 seconds. Out of this time, 27 seconds were spent on getting the database image from the on-disk database in the durability server, sending it through the network, and installing it on the new data server. The remaining time, 159 seconds, was spent receiving the log and applying the updates to the database.

Although the size of the database image ($\approx$ 500 MB) is much larger than that of the log ($\approx$ 6.75 MB), applying the updates takes much longer than loading the image into MySQL. This happens because while applying the updates from the logs is performed by means of SQL statements, loading an image into MySQL can bypass the JDBC interface.

To maximize the effect of the recovery procedure on the normal transaction processing, the experiments were performed while one of the three durability servers was unavailable. However, no meaningful variation on the response time of the operational data servers was verified within this period, showing that the procedure was not disruptive to the system.

How good is recovery in our worst-case analysis? "Five-nines availability" implies about 5 minutes of downtime a year at most. If the whole system became unavailable due to a crash, that would mean in the average a little less than two data server crashes a year if database images are created in 10-minute intervals. In Sprint, however, if a data server crashes, the database becomes only partially unavailable; transactions accessing operational servers can still be executed.

7.4 Micro-benchmark

To evaluate Sprint's performance under a variety of workloads, we conducted experiments using a micro-benchmark. Our micro-benchmark has very simple transactions, containing only two operations. In doing so, we stress communication among servers and reduce local processing on the IMDBs. By varying the ratios of local/global, and read-only/update transactions, we had fine control over the amount of communication induced by each workload.

The database used in the experiments was composed of a single table hash partitioned according to the primary key. Database rows had 100 bytes of application data. There were three attributes in the table: two integers, "id" and "value", where id is the primary key, and a 92-character string, "txt". Read-only transactions were composed of two requests of type: "`SELECT` value `FROM` table `WHERE` id=?"; the id was randomly generated for each SQL statement. Update transactions had a `SELECT` and an update request: "`UPDATE` table `SET` value=? `WHERE` id=?"; both value and id were randomly generated for each SQL statement.

7.4.1 Throughput versus workload

In the experiments that follow, each data server stored a local database with approximately 300 MB of application data (375 MB of physical data). We conducted experiments with 8

DSs, leading to an overall database with 2.4 GB, uniformly divided among data servers. As a reference, we also performed experiments with a standalone MySQL.

The goal of these experiments was to determine the maximum sustainable throughput under different transaction mixes. During the experiments, we made sure that there were enough clients and edge servers to fully load the system. Figure 11 shows the throughput of Sprint for response times of at most 20 milliseconds.

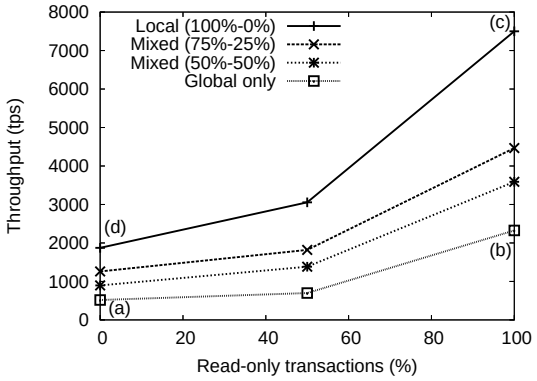

Figure 11: Micro-benchmark throughput

Figure 11 fully characterizes the behavior of Sprint running the micro-benchmark with 8 DSs. Four points in the graph deserve special attention: (a) all transactions are global and update the database, (b) all transactions are global and only read the database, (c) all transactions are local and only read the database, and (d) all transactions are local and update the database. Point (a) requires two multicasts: one to synchronize and another one to terminate transactions. Point (b) has better throughput than (a) because there is a single multicast per transaction and no aborts to ensure the execution order invariant (transactions only read the database). Point (c) provides the best throughput; no multicast is involved. Point (d) requires a single multicast per transaction, to make it durable. As with (b), there are no aborts due to the execution order invariant since transactions are all local.

From point (a) to point (b), the throughput increases 4.5x, and from (a) to (d) it increases 3.6x. The advantage of (b) over (d) is due to the use of two "streams" for multicast, one for synchronizing global transactions and another for terminating update transactions. Since the first is not used to store transaction state, we disabled disk writes—all information was kept in main memory only.

Table 3 shows the ratio between Sprint's throughput and that of a standalone server using a 2.4 GB database. Not surprising, performance of the former is always better than the latter, since the database does not fit in the main memory of the single server. Interestingly, the ratio decreases when we pass from update-only transactions to mixes with 50% of update transactions. This happens because although the throughput increases in Sprint when more transactions are read-only, the improvement is not as big as in a standalone database, and therefore the ratio is smaller. When fewer transactions update the database, a bigger portion of it can be cached in main memory, improving the performance of the standalone server. In Sprint, the effect is not as significant since the database is already all in main memory.

Table 3 depicts in bold configurations in which Sprint running with 8 DSs is at least 8 times

Local Transactions	Read-only Transactions 0%	50%	100%
0%	2.8x	2.0x	5.8x
50%	4.8x	4.0x	**9.0x**
75%	6.8x	5.3x	**11.2x**
100%	**10.1x**	**8.9x**	**18.8x**

Table 3: Throughput improvement w.r.t. single DB

better than a standalone database. In the best mix, when all transactions are local and only read the database, Sprint has a throughput that is more than 18 times that of a standalone database. Two factors account for this: higher parallelism (8 DSs as opposed to a single one) and main memory execution only.

7.4.2 Abort rates

In the following, we consider abort rates under more stringent conditions than TPC-C. Table 4 shows the effects of several transaction mixes on the abort rate. The entries in the table correspond to the experiments depicted in Figure 11. Aborts are highly dependent on the percentage of global transactions in the workload, but not so much dependent on the percentage of update transactions. When all transactions are global and at least half of the them are updates, about 25% of transactions are aborted due to our deadlock avoidance mechanism. Notice that in TPC-C, 85% of transactions are local and 92% are updates.

Local Transactions	Read-only Transactions 0%	50%	100%
0%	0.24471	0.25299	0.00016
50%	0.12098	0.13457	0.00008
75%	0.04762	0.05613	0.00003
100 %	0.00001	0.00001	0.00000

Table 4: Abort rate of micro-benchmark

We have also considered the effects of hot-spots on the deadlock avoidance mechanism. To do so, we changed the micro-benchmark so that one of the transaction operations is forced to access at least one DS in the hot spot area, and varied the size of this area. A hot-spot of size one means that every transaction executes one operation on the designated DS, and so on. The results depicted in Figure 12 are for the worst case scenario, as shown in Table 4, that is, all transactions are global and update the database.

When a single DS is in the hot-spot area, transactions are mostly synchronized by a single data server, and fewer of them abort. With two (or more) DSs in the hot-spot area, local synchronization does not play the same role. Since most transactions access the same DSs, the chances of conflicts increase, and so the aborts. As the number of DSs in the hot-spot area increases, the probability of conflicts decreases together with the aborts, as expected.

7.4.3 Scalability

In the experiments depicted in Figures 13 and 14 we evaluate Sprint's throughput when the number of servers increases from 2 to 32 and the database size is fixed to 500 MB of application

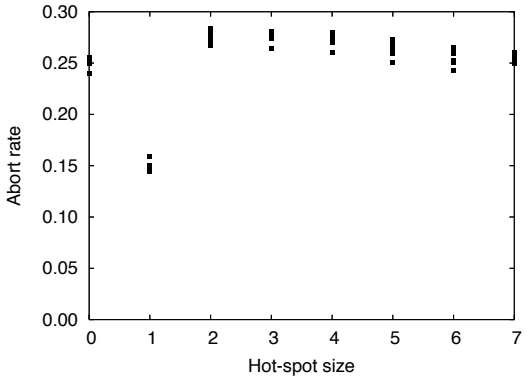

Figure 12: Abort rate with hotspots

data (626 MB of physical data). In all configurations, the data is uniformly distributed among the servers. In both graphs we also show the throughput of our multicast primitive in messages per second (mps) and of a standalone MySQL running databases with 250 MB (physically 314 MB) and 500 MB, corresponding to cases in which the database fits comfortably in the main memory of the single server, and in which it does not. The throughput of the multicast primitive represents a theoretical upper bound on the number of transactions per second that could be achieved in workloads with global or update transactions.

Figure 13 considers a mixed workload with 50% of update transactions—in such cases, total order multicast is always needed to terminate transactions. The reported throughput (in tps) in all cases corresponds to a response time of at most 20 msecs. When compared to MySQL with a 500-MB database, Sprint provides about the same throughput when all transactions are global, and higher throughput when at least 50% of the transactions are local, with peak performance 6.4x bigger than the standalone server when there are 16 DSs. When compared to MySQL running a quite small database of 250 MB, Sprint can perform better if the workload is dominated by local transactions. In such cases, however, the only advantage of Sprint with respect to a single server is fault tolerance: Sprint can quickly recover from the failure of any data server and continue operating with the operational servers while the failed server recovers.

Figure 14 depicts the scalability of the system when all transactions only perform read operations. If transactions are all global, then none of Sprint configurations can process as many transactions per second as a single MySQL server with a database of 500 MB. When 50% of the transactions are local, Sprint performs better for 8, 16 and 32 DSs; in the largest configuration the throughput gets close to the theoretical maximum (i.e., multicast primitive). Finally, when all transactions are local, Sprint scales linearly (notice that the y-axis in the graph is in log scale). MySQL with a database of 250 MB has a throughput near the theoretical maximum that Sprint could achieve. Its performance is clearly much higher than Sprint's, when multicast is needed. If all transactions are local though, Sprint has a throughput 5x higher than MySQL.

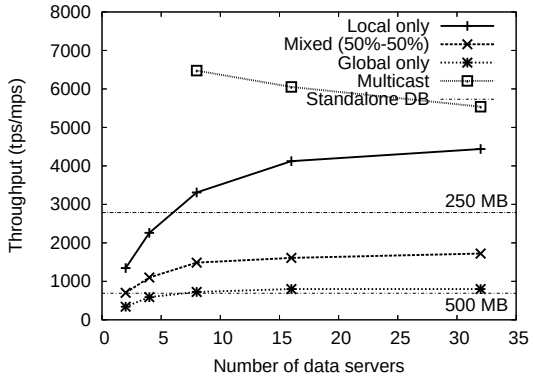

Figure 13: Scalability of mixed load (50% update).

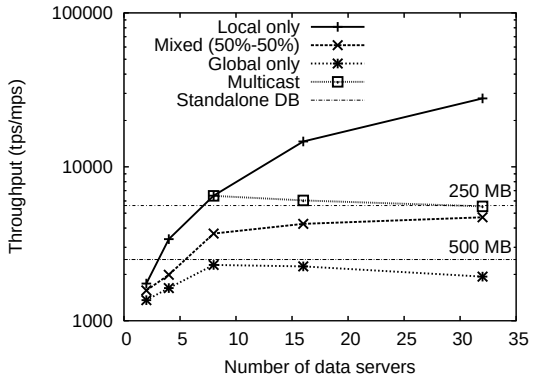

Figure 14: Scalability of read-only transactions.

8 Related work

8.1 On-disk database systems

Increasing the performance and the availability of on-disk databases has been an active area of research for many years. While commercial products (e.g., [20, 34]) have traditionally favored failover based on storage systems shared among the cluster nodes (e.g., disk arrays), research prototypes have targeted clusters of shared-nothing servers (e.g., [14]).

Gray et al. [10] showed that conventional replication algorithms, typically based on distributed locking [3], can lead to high abort rates as a consequence of concurrent accesses. This result motivated much research on protocols based on total-order broadcast, or a variant primitive, to synchronize concurrent accesses and reduce the number of aborts [1, 2, 15, 22, 23, 24, 25, 27]. Some approaches have also considered consistency criteria weaker than serializability. In particular, many recent works have focused on snapshot isolation [5, 6, 18, 26, 29, 33].

Sprint "decouples" replication for availability from replication for performance. Data servers can be replicated for performance only: if a data item is often read and rarely updated, then increasing the number of in-memory replicas of the item will allow more parallel reads; if the item is rarely read but often modified, then performance is improved by reducing its number of in-memory replicas. Data availability is ensured (by the durability servers) even if only a single data server stores the item. Durability servers are used by update transactions at commit time only, and thus, do not interfere with the database synchronization protocol.

How does Sprint compare to on-disk database replication approaches? Middleware-based database replication protocols are mainly for fault tolerance. Depending on the workload and the replication level, usually full replication, throughput can also be improved when replicas are added to the system. Scalability of fully replicated databases, however, is limited to read-intensive workloads. Sprint, on the contrary, can be configured for performance under both read- and write-intensive workloads by judiciously partitioning and replicating the data.

8.2 In-memory database systems

IMDBs were introduced in the early 80's [8] and successfully used in real contexts (e.g., telecommunication industry). To cope with failures, some existing implementations use a primary-backup technique in which the primary propagates system-level information (e.g., database pages) to the backups [7]. Backups monitor the primary and if they suspect it has crashed, some backup takes over. A different approach is proposed in [36] where virtual-memory-mapped communication is used to achieve fast failover by mirroring the primary's memory on the backups.

Some contemporary systems have considered IMDBs in clustered environments. MySQL Cluster [28] replicates and fragments the database space among server clusters to enhance performance and availability. To ensure good performance of update transactions as well, the approach makes use of *deferred disk writes*. This means that updates are written to disk after the transaction has committed. Transaction atomicity is ensured by synchronizing the servers' disk writes, but some failure patterns may violate the durability property. An alternative approach to keeping a partitioned database consistent while relaxing durability is discussed in [30].

The work in [13] consists in interposing an IMDB between applications and on-disk databases as a content cache. The database is partitioned across individual servers in such a way that

queries can be executed in a single database without data transmission or synchronization with other servers. Sprint could benefit from such a partitioning scheme in order to reduce the number of global transactions.

9 Final remarks

Sprint's distributed data management protocol was designed to stress processors, main memories, and the network, while sparing disk. No restrictive assumptions are made about the failure model (e.g., no server must be always up) and failure detection. When disk access is unavoidable, it is done sequentially. The execution model is very simple, favoring local transactions. An implicit assumption of the approach is that by carefully partitioning a database, many transactions can be made local, maximizing performance.

Performance experiments have demonstrated that for a large range of workloads, Sprint can extend the functionality of a single-server IMDB to a cluster of such servers. It shows that middleware architectures can be also employed to design highly efficient and fault-tolerant data management systems even when the database is not fully replicated. We hope it will open up new directions in research on high performance and high availability middleware-based database protocols.

Acknowledgments

We would like to thank Prof. Bettina Kemme and the anonymous reviewers for their suggestions to improve the presentation of this work, and Apple Computer International and Emulab for the hardware infrastructure, which allowed us to develop, fine tune, and evaluate Sprint.

References

[1] D. Agrawal, G. Alonso, A. E. Abbadi, and I. Stanoi. Exploiting atomic broadcast in replicated databases. In *Proceedings of EuroPar (EuroPar'97)*, Passau (Germany), 1997.

[2] Y. Amir and C. Tutu. From total order to database replication. In *International Conference on Distributed Computing Systems (ICDCS)*, July 2002.

[3] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[4] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[5] S. Elnikety, S. Dropsho, and F. Pedone. Tashkent: Uniting durability with transaction ordering for high-performance scalable database replication. In *Proceedings of EuroSys*, 2006.

[6] S. Elnikety, F. Pedone, and W. Zwaenepoel. Database replication using generalized snapshot isolation. In *Symposium on Reliable Distributed Systems (SRDS'2005)*, Orlando, USA, 2005.

[7] FirstSQL Inc. The FirstSQL/J in-memory database system. http://www.firstsql.com.

[8] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, 1992.

[9] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, 2006.

[10] J. N. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, Montreal (Canada), 1996.

[11] J. N. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[12] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In *Distributed Systems*, chapter 5. Addison-Wesley, 2nd edition, 1993.

[13] M. Ji. Affinity-based management of main memory database clusters. *ACM Transactions on Internet Technology (TOIT)*, 2(4):307–339, 2002.

[14] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In *Proceedings of 26th International Conference on Very Large Data Bases (VLDB'2000)*, Cairo, Egypt, 2000.

[15] B. Kemme and G. Alonso. A new approach ro developing and implementing eager database replication protocols. *ACM Transactions on Database Systems (TODS)*, 25(3), September 2000.

[16] D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, 2000.

[17] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[18] Y. Lin, B. Kemme, M. Patino-Martinez, and R. Jimenez-Peris. Middleware based data replication providing snapshot isolation. In *International Conference on Management of Data (SIGMOD)*, Baltimore, Maryland, USA, 2005.

[19] D. Morse. In-memory database web server. *Dedicated Systems Magazine*, (4):12–14, 2000.

[20] Oracle parallel server for windows NT clusters. Online White Paper.

[21] M. T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1999.

[22] M. Patino-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Scalable replication in database clusters. In *Disctributed Computing (DISC)*, 2000.

[23] M. Patino-Martinez, R. Jimenez-Peris, B. Kemme, and G. Alonso. Consistent database replication at the middleware level. *ACM Transactions on Computer Systems*, 23(4):375–423, 2005.

[24] F. Pedone and S. Frølund. Pronto: A fast failover protocol for off-the-shelf commercial databases. In *Proceedings of* 19^{th} *IEEE Symposium on Reliable Distributed Systems (SRDS'2000)*, Nürnberg, Germany, 2000.

[25] F. Pedone, R. Guerraoui, and A. Schiper. Transaction reordering in replicated databases. In *Proceedings of the 16th IEEE Symposium on Reliable Distributed Systems*, Durham (USA), 1997.

[26] C. Plattner and G. Alonso. Ganymed: scalable replication for transactional web applications. In *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pages 155–174, 2004.

[27] L. Rodrigues, H. Miranda, R. Almeida, J. Martins, and P. Vicente. Strong replication in the GLOBDATA middleware. In *Workshop on Dependable Middleware-Based Systems*, 2002.

[28] M. Ronström and L. Thalmann. Mysql cluster architecture overview. MySQL Technical White Paper, 2004.

[29] R. Schenkel, G. Weikum, N. Weissenberg, and X. Wu. Federated transaction management with snapshot isolation. In *Selected papers from the Eight International Workshop on Foundations of Models and Languages for Data and Objects, Transactions and Database Dynamics*, pages 1–25, 2000.

[30] R. Schmidt and F. Pedone. Consistent main-memory database federations under deferred disk writes. In *Symposium on Reliable Distributed Systems (SRDS'2005)*, Orlando, USA, 2005.

[31] T. Shetler. In-memory databases: The catalyst behind real-time trading systems. http://www.timesten.com/library/.

[32] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, 2002. USENIX Association.

[33] S. Wu and B. Kemme. Postgres-r(si): Combining replica control with concurrency control based on snapshot isolation. In *Proceedings of the International Conference of Data Engineering*, 2005.

[34] Informix extended parallel server 8.3. Online White-Paper.

[35] C. Yu and W. Meng. *Principles of Database Query Processing for Advanced Applications*. Morgan Kaufmann, San Francisco, 1998.

[36] Y. Zhou, P. Chen, and K. Li. Fast cluster failover using virtual memory-mapped communication. Technical Report TR-591-99, Princeton University, 1999.

Appendix

In the following we prove two properties of the algorithm: (a) Sprint's data management protocol ensures strong consistency, that is, all executions are one-copy serializable, and (b) Sprint detects and solves distributed deadlocks.

We initially introduce a simple formalism. A history h over a set of transactions $\mathcal{T} = \{T_1, T_2, ..., T_n\}$ is a partial order $\prec$ where (a) h contains the operations of each transaction in $\mathcal{T}$; (b) for each $T_i \in \mathcal{T}$, and all operations O_i and O_i' in T_i: if O_i precedes O_i' in T_i, then $O_i \prec O_i'$ in h; and (c) if T_i reads X from T_j, then $W_j(X_j) \prec R_i(X_j)$ in h [3], where $R_i(X_j)$ (resp., $W_i(X_i)$) is a read (write) operation performed by T_i over data item X_j (X_i).

Proposition 1 *Sprint's data management ensures one-copy serializability.*

PROOF: We show that every history h produced by Sprint has an acyclic multi-version serialization graph (MVSG). From [3], if $MVSG(h)$ is acyclic, then h is view equivalent to some serial execution of the same transactions using a single-copy database. MVSG is a directed graph with the nodes representing committed transactions and edges representing dependencies between them. There are three types of directed edges in MVSG: (a) read-from edges, (b) version-order edges type I, and (c) version-order edges type II.

From the algorithm, the commit order of transactions induces a version order on every data item: If $\ll$ is an order relation on the versions, and T_i and T_j update X, then we have $commit(T_i) < commit(T_j) \Leftrightarrow X_i \ll X_j$. To show that $MVSG(h)$ has no cycles, we prove that for every edge $T_i \rightarrow T_j$ in $MVSG(h)$, we have $commit(T_i) < commit(T_j)$. The proof continues by considering each edge type in $MVSG(h)$.

1. Read-from edge – If T_j reads data item X_i from T_i (i.e., $R_j(X_i)$), then $T_i \rightarrow T_j \in MVSG(h)$.
   We have to show that $commit(T_i) < commit(T_j)$. This follows from the fact that since T_i is an update transaction, it executes in isolation. Other transactions can only read X_i after T_i has committed.

2. Version-order edge type I. If both T_i and T_j write X such that $X_i \ll X_j$, then $T_i \rightarrow T_j \in MVSG(h)$. Since the commit order induces the version order, we have that $X_i \ll X_j \Leftrightarrow commit(T_i) < commit(T_j)$.

3. Version-order edge type II. If T_i reads X_k from T_k, and both T_k and T_j write X such that $X_k \ll X_j$, then $T_i \rightarrow T_j \in MVSG(h)$. We have to show that $commit(T_i) < commit(T_j)$. Assume that T_i reads X_k from data server DS. Since T_j is an update transaction, it can only commit if it modifies all data servers storing a copy of X. Consider first an execution without failures. Since T_i reads T_k's updates, it access DS after T_k's commit and before any other update transaction. Therefore, T_j will be blocked at DS waiting for T_i to finish and release the lock DS. It follows that $commit(T_i) < commit(T_j)$. Consider now the case in which T_i reads X_k at DS, this one fails, another instance of it, say DS', is created, and T_j updates DS'. Since T_i commits, before failing, DS must have voted to commit T_i. This vote is totally ordered with the message m that informed about the replacement of DS by DS'. Had DS's vote been delivered after m, T_i would have been aborted. Therefore, DS's vote must have been delivered before m, and T_i was committed before T_j accessed DS'. Hence, $commit(T_i) < commit(T_j)$. □

Proposition 2 *If a transaction is involved in a deadlock then it is aborted.*

PROOF: From the multiple-readers single-writer policy, deadlocks involving local transactions cannot happen. We show that deadlocks involving global transactions are avoided. From the algorithm invariant (see Section 4.1), a global transaction with a lower sequential number never waits for a conflicting global transaction with a higher sequential number. Thus, it remains to be proved that for any pair of transactions T_i and T_j, if $seq(T_i) < seq(T_j)$ at data server DS_k, then $seq(T_i) < seq(T_j)$ at data server DS_l. In the absence of failures this trivially follows from the total order property of total order multicast and the way sequential numbers are assigned by the algorithm. Consider now that DS_l assigns a sequential number to T_i, fails, and upon recovering assigns a smaller sequential number to T_j. This can only happen if the message informing that T_i became global is received by DS_l before this one fails. Thus, the new instance of DS_l, after the failure, will not have a sequential number for T_i. Upon receiving the first request for T_i, DS_l will abort T_i. □