USENIX Association

Proceedings of the FREENIX Track: 2004 USENIX Annual Technical Conference

Boston, MA, USA
June 27–July 2, 2004

C-JDBC: Flexible Database Clustering Middleware

Emmanuel Cecchet
INRIA Rhône-Alpes
emmanuel.cecchet@inria.fr

Julie Marguerite
ObjectWeb Consortium
julie.marguerite@objectweb.org

Willy Zwaenepoel
EPF Lausanne
willy.zwaenepoel@epfl.ch

Abstract

Large web or e-commerce sites are frequently hosted on clusters. Successful open-source tools exist for clustering the front tiers of such sites (web servers and application servers). No comparable success has been achieved for scaling the backend databases. An expensive SMP machine is required if the database tier becomes the bottleneck. The few tools that exist for clustering databases are often database-specific and/or proprietary.

Clustered JDBC (C-JDBC) addresses this problem. It is a freely available, open-source, flexible and efficient middleware for database clustering. C-JDBC presents a single virtual database to the application through the JDBC interface. It does not require any modification to JDBC-based applications. It furthermore works with any database engine that provides a JDBC driver, without modification to the database engine. The flexible architecture of C-JDBC supports large and complex database cluster architectures offering various performance, fault tolerance and availability tradeoffs.

We present the design and the implementation of C-JDBC, as well as some uses of the system in various scenarios. Finally, performance measurements using a clustered implementation of the TPC-W benchmark show the efficiency and scalability of C-JDBC.

1. Introduction

Database scalability and high availability can be achieved in the current state-of-the-art, but only at very high expense. Existing solutions require large SMP machines and high-end RDBMS (Relational Database Management Systems). The cost of these solutions, both in terms of hardware prices and software license fees, makes them available only to large businesses.

Clusters of commodity machines have largely replaced large SMP machines for scientific computing because of their superior price/performance ratio. Clusters are also used to provide both scalability and high availability in data server environments. This approach has been successfully demonstrated, for instance, for web servers and for application servers [9]. Success has been much more limited for databases. Although there has been a large body of research on replicating databases for scalability and availability [3], very few of the proposed techniques have found their way into practice [17].

Recently, commercial solutions such as Oracle Real Application Clusters [16] have started to address cluster architectures using a shared storage system such as a SAN (Storage Area Network). The IBM DB2 Integrated Cluster Environment [5] also uses a shared storage network to achieve both fault tolerance and performance scalability.

Open-source solutions for database clustering have been database-specific. MySQL replication [14] uses a master-slave mechanism with limited support for transactions and scalability. Some experiments have been reported using partial replication in Postgres-R [13]. These extensions to existing database engines often require applications to use additional APIs to benefit from the clustering features. Moreover, these different implementations do not interoperate well with each other.

We present Clustered JDBC (C-JDBC), an open-source middleware solution for database clustering on a shared-nothing architecture built with commodity hardware. C-JDBC hides the complexity of the cluster and offers a single database view to the application. The client application does not need to be modified and transparently accesses a database cluster as if it were a centralized database. C-JDBC works with any RDBMS that provides a JDBC driver. The RDBMS does not need any modification either, nor does it need to provide distributed database functionalities. Load distribution, fault tolerance and failure recovery are all handled by C-JDBC. The architecture is flexible and can be distributed to support large clusters of heterogeneous databases with various degrees of performance, fault tolerance and availability.

With C-JDBC we hope to make database clustering available in a low-cost and powerful manner, thereby spurring its use in research and industry. Although C-JDBC has only been available for a few months, several installations are already using it to support various database clustering applications.

The outline of the rest of this paper is as follows. Section 2 presents the architecture of C-JDBC and the role of each of its components. Section 3 describes how fault tolerance is handled in C-JDBC. Section 4 discusses horizontal and vertical scalability. Section 5 documents some uses of C-JDBC. Section 6 describes measurement results using a clustered implementation of the TPC-W benchmark. We conclude in Section 7.

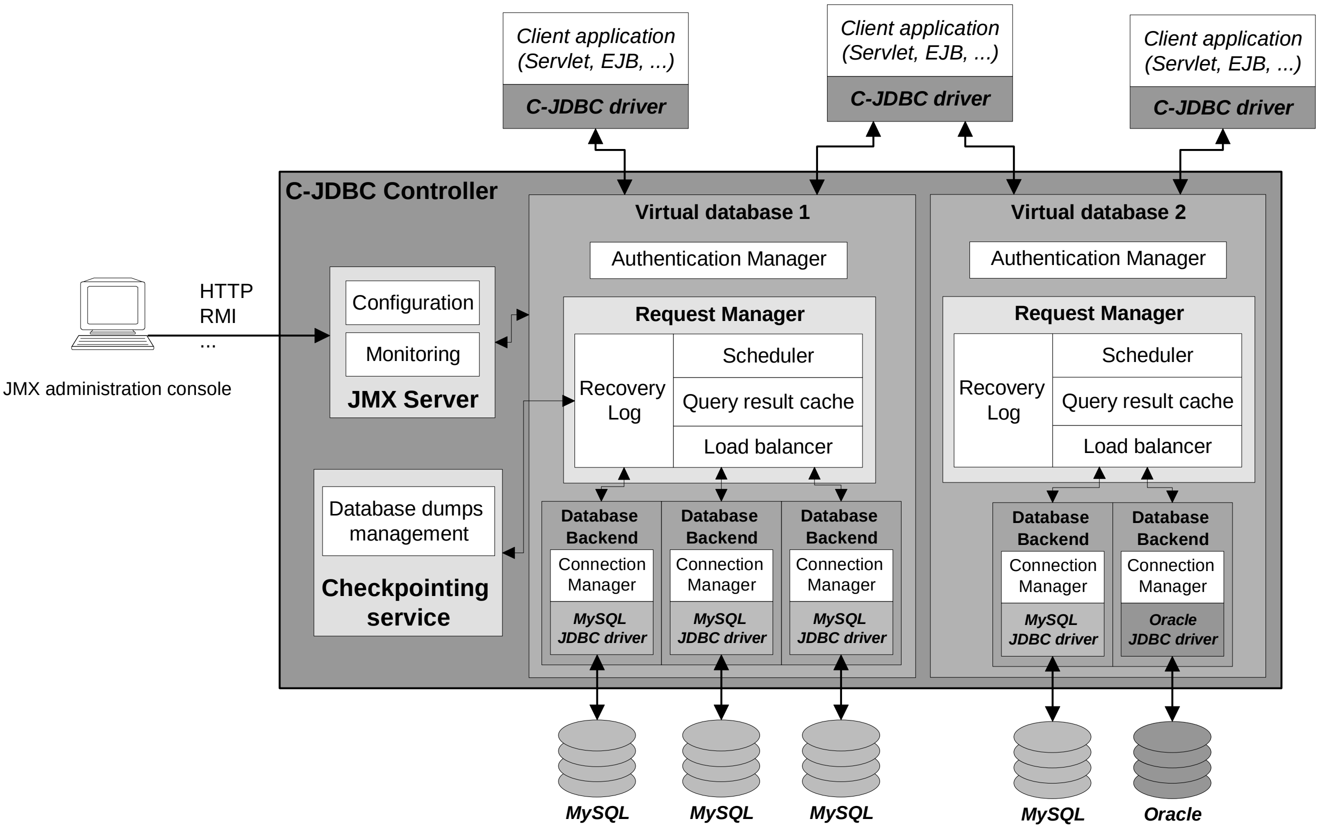


Figure 1. C-JDBC architecture overview

2. C-JDBC architecture

This section provides a functional overview of C-JDBC, its overall architecture, and its key components.

2.1. Functional overview

JDBC™ is a Java API for accessing virtually any kind of tabular data [19]. C-JDBC (Clustered JDBC) is a Java middleware for database clustering based on JDBC. It turns a collection of possibly heterogeneous databases into a single virtual database. No changes are needed to the application or to the databases.

Various data distributions are supported: the data can either be fully replicated, partially replicated, or partitioned, depending on the needs of the application. The degree of replication and the location of the replicas can be specified on a per-table basis. Currently, the tables named in a particular query must all be present on at least one backend. In dynamic content servers, one of the target environments for C-JDBC clusters, this requirement can often be met, since the queries are known ahead of time. Eventually, we plan to add support for distributed execution of a single query.

Routing of queries to the various backends is done automatically by C-JDBC, using a read-one write-all approach. A number of strategies are available for load balancing, with the possibility of overriding these strategies with a user-defined one. C-JDBC can be configured to support query response caching and fault tolerance. Finally, larger and more highly-available systems can be built by suitable combinations of individual C-JDBC instances.

C-JDBC also provides additional services such as monitoring and logging. The controller can be dynamically configured and monitored through JMX (Java Management eXtensions) either programmatically or using an administration console.

2.2. Architecture

Figure 1 gives an overview of the different C-JDBC components. The client application uses a C-JDBC driver that replaces the database-specific JDBC driver but offers the same interface. The C-JDBC controller is a Java program that acts as a proxy between the C-JDBC driver and the database backends. The controller exposes a single database view, called a *virtual database*, to the C-JDBC driver and thus to the application. A controller can host multiple virtual databases, as shown in the figure. Each virtual database has its own request manager that defines its request scheduling, caching and load balancing policies.

The database backends are accessed through their native JDBC driver. If the native driver is not capable of connection pooling, C-JDBC can be configured to provide a connection manager for this purpose.

In the rest of this section we discuss the key components of the C-JDBC architecture, in particular the driver and the request manager. Other components that provide standard functionality such as authentication management, connection management and configuration support are not discussed further. More detail on these components and other aspects of C-JDBC can be found at http://c-jdbc.objectweb.org.

2.3. C-JDBC driver

The C-JDBC driver is a hybrid type 3 and type 4 JDBC driver[19]. It implements the JDBC 2.0 specification and some extensions of the JDBC 3.0 specification. All processing that can be performed locally is implemented inside the C-JDBC driver. For example, when an SQL statement has been executed on a database backend, the result set is serialized into a C-JDBC driver ResultSet that contains the logic to process the results. Once the ResultSet is sent back to the driver, the client can browse the results locally.

All database-dependent calls are forwarded to the C-JDBC controller that issues them to the database native driver. The native database driver is a type 3 JDBC driver. SQL statement executions are the only calls that are completely forwarded to the backend databases. Most of the C-JDBC driver remote calls can be resolved by the C-JDBC controller itself without going to the databases.

The C-JDBC driver can also transparently fail over between multiple C-JDBC controllers, implementing horizontal scalability (see Section 4).

2.4. Request manager

The request manager contains the core functionality of the C-JDBC controller. It is composed of a scheduler, a load balancer and two optional components: a recovery log and a query result cache. Each of these components can be superseded by a user-specified implementation.

2.4.1. Scheduler

When a request arrives from a C-JDBC driver, it is routed to the request manager associated with the virtual database. Begin transaction, commit and abort operations are sent to all backends. Reads are sent to a single backend. Updates are sent to all backends where the affected tables reside. Depending on whether full or partial replication is used (see Section 2.4.3), this may be one, several or all backends. SQL queries containing macros such as RAND() or NOW() are rewritten on-the-fly with a value computed by the scheduler so that each backend stores exactly the same data.

All operations are synchronous with respect to the client. The request manager waits until it has received responses from all backends involved in the operation before it returns a response to the client.

If a backend executing an update, a commit or an abort fails, it is disabled. In particular, C-JDBC does not use a 2-phase commit protocol. Instead, it provides tools to automatically re-integrate failed backends into a virtual database (see Section 3).

At any given time only a single update, commit or abort is in progress on a particular virtual database. Multiple reads from different transactions can be going on at the same time. Updates, commits and aborts are sent to all backends in the same order.

2.4.2. Query result cache

An optional *query result cache* can be used to store the result set associated with each query. The query result cache reduces the request response time as well as the load on the database backends. By default, the cache provides strong consistency. In other words, C-JDBC invalidates cache entries that may contain stale data as a result of an update query. Cache consistency may be relaxed using user-defined rules. The results of queries that can accept stale data can be kept in the cache for a time specified by a staleness limit, even though subsequent update queries may have rendered the cached entry inconsistent.

We have implemented different cache invalidation granularities ranging from database-wide invalidation to table-based or column-based invalidation with various optimizations.

2.4.3. Load balancer

If no cache has been loaded or a cache miss has occurred, the request arrives at the *load balancer*.

C-JDBC offers various load balancers according to the degree of replication the user wants. Full replication is easy to handle. It does not require request parsing since every database backend can handle any query. Database updates, however, need to be sent to all nodes, and performance suffers from the need to broadcast updates when the number of backends increases.

To address this problem, C-JDBC provides partial replication in which the user can define database replication on a per-table basis. Load balancers supporting partial replication must parse the incoming queries and need to know the database schema of each backend. The schema information is dynamically gathered. When a backend is enabled, the appropriate methods are called on the JDBC DatabaseMetaData information of the backend native driver. Database schemas can also be specified statically by way of a configuration file. The schema is updated dynamically on each *create* or *drop* SQL statement to accurately reflect each backend schema.

Among the backends that can treat a read request (all of them with full replication), one is selected according to the load balancing algorithm. Currently implemented algorithms are round robin, weighted round robin and least pending requests first (the request is sent to the node that has the least pending queries).

2.4.4. Optimizations

To improve performance, C-JDBC implements *parallel transactions*, *early response to update, commit, or abort requests*, and *lazy transaction begin*.

With parallel transactions, operations from different transactions can execute at the same time on different backends. Early response to update, commit or abort allows the controller to return the result to the client application as soon as one, a majority or all backends have executed the operation. Returning the result when the first backend completes the command offers the latency of the fastest backend to the application. When early response to update is enabled, C-JDBC makes sure that the order of operations in a single transaction is respected at all backends. Specifically, if a read follows an update in the same transaction, that read is guaranteed to execute after the update has executed.

Finally, with lazy transaction begin, a transaction is started on a particular backend only when that backend needs to execute a query for this transaction. An update query on a fully replicated cluster causes a transaction to be started on all backends. In contrast, for read-only transactions, a transaction is started only on the backend that executes the read queries of the transaction. On a read-mostly workload this optimization significantly reduces the number of transactions that need to be initiated by an individual backend.

3. Fault tolerance

C-JDBC provides checkpoints and a recovery log to allow a backend to restart after a failure or to bring new backends into the system.

3.1. Checkpointing

A checkpoint of a virtual database can be performed at any point in time. Checkpointing can be manually triggered by the administrator or automated based on temporal rules.

Taking a snapshot of a backend while the system is online requires disabling this backend so that no updates occur on it during the backup. The other backends remain enabled to answer client requests. As the different backends of a virtual database need to remain consistent, backing up a backend while leaving it enabled would require locking all tables in read mode and thus blocking all updates on all backends. This is not possible when dealing with large databases where copying the database content may take hours.

The checkpoint procedure starts by inserting a checkpoint marker in the recovery log (see Section 3.2). Next, the database content is dumped. Then, the updates that occurred during the dump are replayed from the recovery log to the backend, starting at the checkpoint marker. Once all updates have been replayed, the backend is enabled again.

C-JDBC uses an ETL (Extraction Transforming Loading) tool called Octopus [10] to copy data to or from databases. The database (including data and metadata) is stored in a portable format. Octopus re-creates the tables and the indexes using the database-specific types and syntax.

3.2. Recovery log

C-JDBC implements a recovery log that records a log entry for each begin, commit, abort and update statement. A log entry consists of the user identification, the transaction identifier, and the SQL statement. The log can be stored in a flat file, but also in a database using JDBC. A fault-tolerant log can then be created by sending the log updates to a virtual C-JDBC database with fault tolerance enabled. Figure 2 shows an example of a fault-tolerant recovery log. The log records are sent to a virtual database inside the same C-JDBC controller as the application database, but this virtual database could have been hosted on a different controller as well. Backends used to store the log can be shared with those used for the application virtual database, or separate backends can be used.

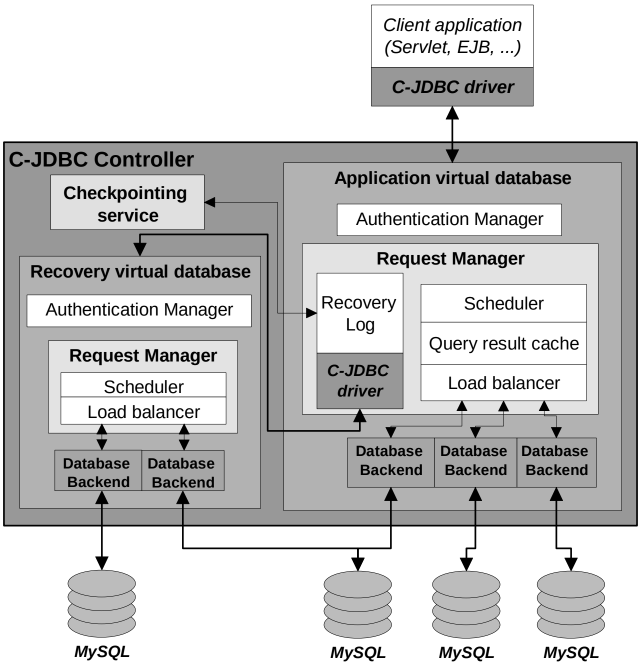


Figure 2. Fault tolerant recovery log example

4. Horizontal and vertical scalability

The C-JDBC controller is potentially a single point of failure. *Horizontal scalability* reduces the probability of system failure by replicating the C-JDBC controller. To support a large number of database backends, we also provide *vertical scalability* to build a hierarchy of backends.

4.1. C-JDBC horizontal scalability

Horizontal scalability prevents the C-JDBC controller from becoming a single point of failure. We use the JGroups [2] group communication library to synchronize the schedulers of the virtual databases that are distributed over several controllers. Figure 3 gives an overview of the C-JDBC controller horizontal scalability.

When a virtual database is loaded in a controller, a group name can be assigned to the virtual database. This group name is used to communicate with other controllers hosting the same virtual database. At initialization time, the controllers exchange their respective backend configurations. If a controller fails, a remote controller can recover the backends of the failed controller using the information gathered at initialization time.

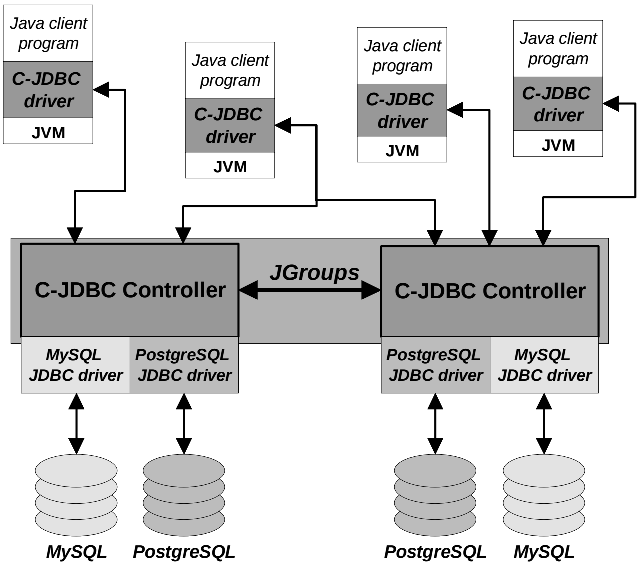


Figure 3. C-JDBC horizontal scalability

C-JDBC relies on JGroups' reliable and ordered message delivery to synchronize write requests and demarcate transactions. Only the request managers contain the distribution logic and use group communication. All other C-JDBC components (scheduler, cache, and load balancer) remain the same.

4.2. C-JDBC vertical scalability

It is possible to nest C-JDBC controllers by re-injecting the C-JDBC driver into the C-JDBC controller. Figure 4 illustrates an example of a 2-level C-JDBC composition.

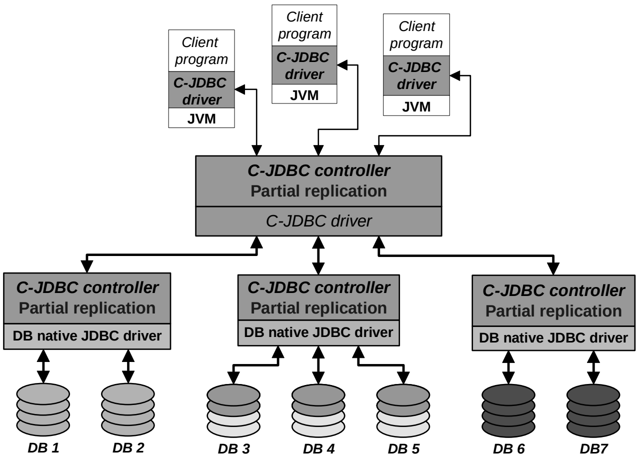


Figure 4. C-JDBC vertical scalability

The top-level controller has been configured for partial replication with three database backends that are virtual databases implemented by other C-JDBC controllers. The C-JDBC driver is used as the backend native driver to access the underlying controller. In general, an arbitrary tree structure can be created. The C-JDBC controllers at the different levels are interconnected by C-JDBC drivers. The native database drivers connect the leaves of the controller hierarchy to the real database backends.

Vertical scalability may be necessary to scale an installation to a large number of backends. Limitations in current JVMs restrict the number of outgoing connections from a C-JDBC driver to a few hundreds. Beyond that, performance drops off considerably. Vertical scalability spreads the number of connections over a number of JVMs, retaining good performance.

4.3. Mixing horizontal and vertical scalability

To deal with very large configurations where both high availability and high performance are needed, one can combine horizontal and vertical scalability. Figure 5 shows an example of such a configuration. The top-level C-JDBC controllers use horizontal scalability for improved availability. Additional controllers are cascaded to provide performance by way of vertical scalability.

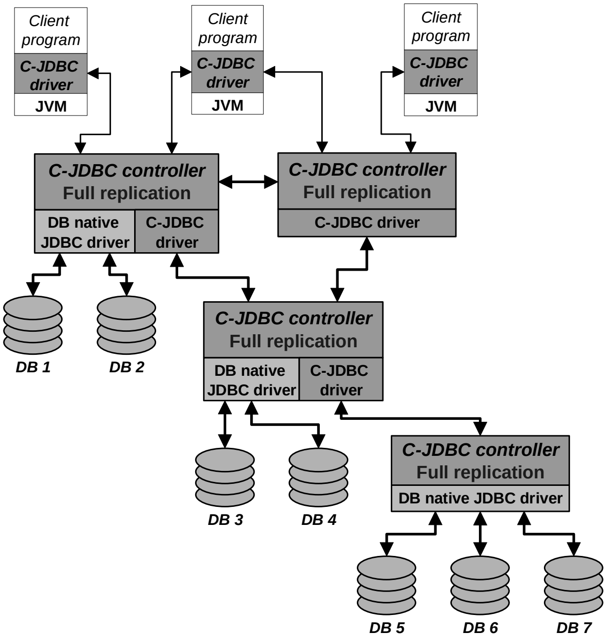
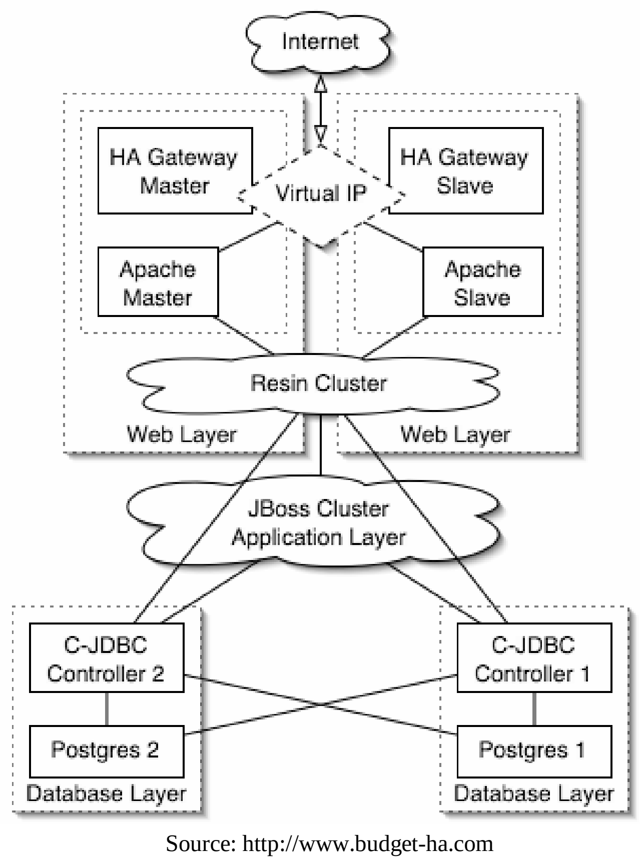


Figure 5. Cascading C-JDBC controllers

In this configuration the top-level controllers would normally be configured with early response to updates and commits (see Section 2.4.4) so that the updates can propagate asynchronously down the tree. Read queries are initially sent to the backends DB1 and DB2 that are connected directly to the top-level controller. Once they become loaded, the load balancer in the top-level controller starts to send queries to the lower–level controller, to be executed by backends DB3 and DB4. The more load the system receives, the deeper in the tree the requests go. In this example, the failure of the middle-level controller makes DB5, DB6 and DB7 unavailable. To remedy this situation, one could use horizontal scalability to replicate the middle-level controller.

5. Sample uses of C-JDBC

Although C-JDBC has only recently been made available, we have already seen considerable use by others. C-JDBC users have different interest and usage scenarios. The flexibility of C-JDBC permits tuning the system for a variety of needs. We present four different use cases focusing on different sets of features. The first example shows how to build a low-cost highly-available system. The next use case demonstrates the combination of portability, performance scalability and high availability for a large production environment with thousands of users and a wide range of operating systems and database backends. The third scenario features a flood alert system where C-JDBC is used to support fast disaster recovery. Finally, the last use case focuses on performance and explains how C-JDBC is used to benchmark J2EE clusters.



Source: http://www.budget-ha.com

Figure 6. Budget high availability solution from budget-ha.com

5.1. Budget High Availability

Our experience indicates that most C-JDBC users are interested in clustering primarily to provide high availability. Their goal is to eliminate every single point of failure in the system. Budget-HA.com [6] has built a solution from open-source components providing a high-availability infrastructure “on a budget”. Figure 6 gives an overview of the proposed 3-tier J2EE infrastructure.

The high availability of the web tier combines Linux-HA with a cluster of Resin servlet containers [7]. The JBoss J2EE server provides clustering features to ensure high availability of the business logic in the application tier. The database tier uses C-JDBC with full replication on two PostgreSQL backends to tolerate the failure of a backend.

The C-JDBC controller is also replicated. Both controllers share the two PostgreSQL backends so that the failure of one controller does not make the system unavailable. In this configuration, the system survives the failure of any component. The minimum hardware configuration requires 2 nodes, each of them hosting an instance of Resin, JBoss, a C-JDBC controller and a PostgreSQL backend.

5.2. OpenUSS: University Support System

OpenUSS [15] provides an open-source system for managing computer assisted learning and computer assisted teaching applications. There are currently 11 universities in Europe, Indonesia and Mexico using OpenUSS.

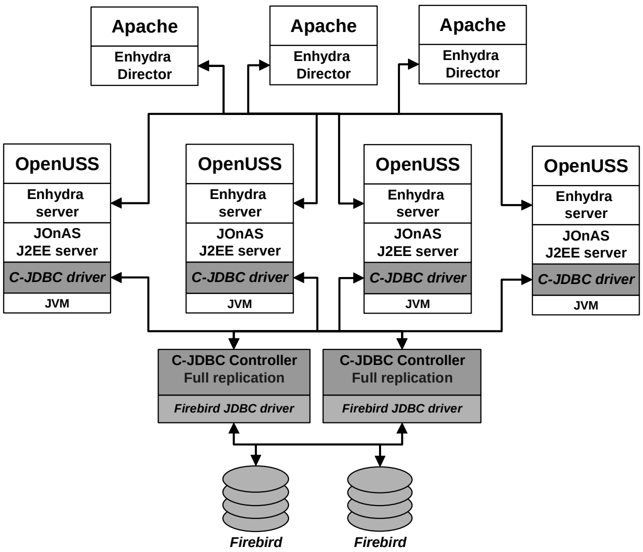


Figure 7. OpenUSS setup at University of Muenster

The largest OpenUSS site runs at University of Muenster in Germany. The system manages more than 12,000 students and over 1,000 instructors. The average workload consists of 180,000 to 200,000 accesses per day to web pages that are dynamically generated from data stored in the database. Lecture materials (PDF documents, slides, etc.) are also stored in the database in the form of BLOBs (Binary Large Objects).

To provide both performance scalability and high availability, C-JDBC is used to replicate the Firebird database backends. Figure 7 shows the C-JDBC configuration used at University of Muenster to run OpenUSS.

Three Apache servers function as frontends. Enhydra Director is used as an Apache module to load balance the queries on four Enhydra [9] servers hosting the OpenUSS application. Each Enhydra server relies on a JOnAS J2EE server to access the database tier. All servers are using the C-JDBC driver to access a replicated C-JDBC controller hosting a fully replicated Firebird database.

The operating systems in use at the universities using OpenUSS include Linux, HP-UX and Windows. The database engines include InterBase, Firebird, PostgreSQL and HypersonicSQL. As C-JDBC is written in Java and does not require any application or database changes, it accommodates all of these environments.

5.3. Flood alert system

floodalert.org is implementing a replacement for a flood alert system for Rice University and the Texas Medical Center. Geographic distribution in this system is essential, because the system must continue to perform if the two sites are threatened by a flood.

JBoss is used for application level clustering, and C-JDBC provides database clustering of MySQL databases. All nodes are located on a VPN to deal with the security issues resulting from running a cluster over a public network. Horizontal scalability with transparent failover is the most important C-JDBC feature, because the system has to be able to survive the loss of any node at any time. The ability to have database vendor independence was also much appreciated in this project.

Figure 8 gives an overview of the floodalert.org system. There are at least three nodes in the system at all times, each with its own database and application server, and each at a different site. At least one site is several hundred miles from the others for disaster recovery.

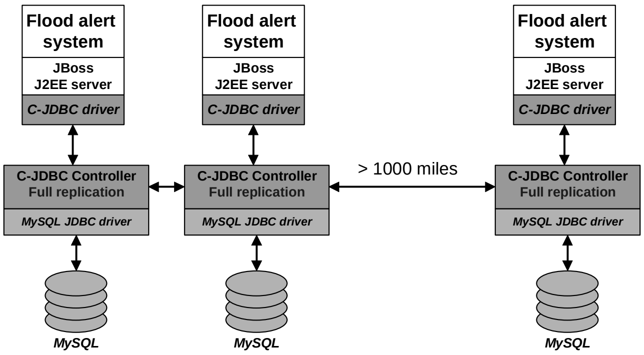


Figure 8. Flood alert system using C-JDBC

C-JDBC's ability to dynamically add and remove nodes allows floodalert.org to bring nodes, either new or stale, into the system without much work. Future versions of the system may include bootable CD-ROMs (like Knoppix or Gentoo LiveCD) that will allow floodalert.org to quickly add a node to the system from any computer with an internet connection.

5.4. J2EE cluster benchmarking

JMOB (Java Middleware Open Benchmarking) [12] is an ObjectWeb initiative for benchmarking middleware. When running J2EE benchmarks on commodity hardware, the database is frequently the bottleneck resource [8]. Therefore, increased database performance is crucial in order to observe the scalability of the J2EE server. Figure 9 shows a J2EE cluster benchmarking environment example.

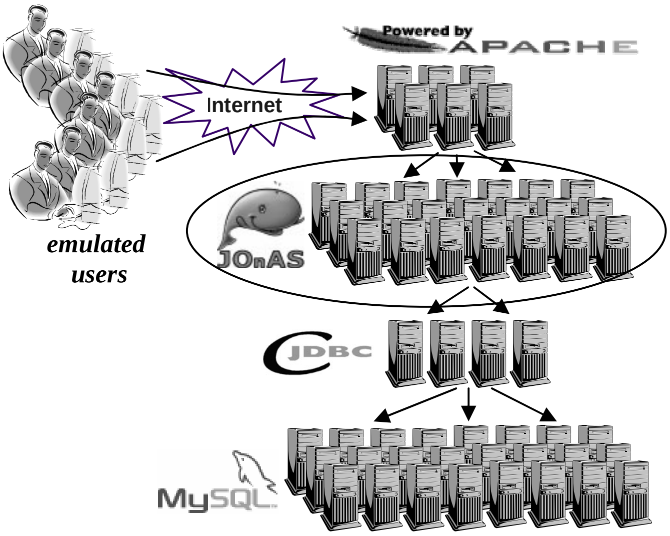


Figure 9. J2EE cluster benchmarking environment

A number of emulated users send HTTP requests to a cluster of web servers (Apache in this example). The Apache servers forward the requests to the J2EE cluster under test (JOnAS in this example). The JOnAS cluster accesses a single virtual database that is implemented by a C-JDBC cluster using several controllers and a large number of backends to scale up to the load required by the J2EE cluster.

The vertical scalability and support for partial replication allows large scale configurations providing high performance and sustained throughput. The next section describes a performance evaluation of C-JDBC.

6. Performance evaluation

To provide some indication of the performance and the scalability of clusters built using C-JDBC, we describe next a set of experiments carried out with a clustered implementation of the TPC-W benchmark. We also show some results for query response caching using the Rubis benchmark.

6.1. Experimental environment

The Web server is Apache v.1.3.22, and Jakarta Tomcat v.4.1.27 [11] is used as the servlet server. We use MySQL v.4.0.12 [14] as our database server with the InnoDB transactional tables and the MM-MySQL v2.0.14 type 4 JDBC driver. The Java Virtual Machine used for all experiments is

IBM JDK 1.3.1 for Linux. All machines run the 2.4.16 Linux kernel.

We use up to six database backends. Each machine has two PII-450 MHz CPUs with 512MB RAM, and a 9GB SCSI disk drive[1]. In our evaluation, we are not interested in the absolute performance values but rather by the relative performance of each configuration. Having slower machines allows us to reach the bottlenecks without requiring a large number of client machines to generate the necessary load. All machines are connected through a switched 100Mbps Ethernet LAN.

6.2. The TPC-W benchmark

The TPC-W specification [18] defines a transactional Web benchmark for evaluating e-commerce systems. TPC-W simulates an online bookstore like amazon.com.

Of the 14 interactions specified in the TPC-W benchmark specification, six are read-only and eight have update queries that change the database state. TPC-W specifies three different workload mixes, differing in the ratio of read-only to read-write interactions. The *browsing* mix contains 95% read-only interactions, the *shopping* mix 80%, and the *ordering* mix 50%. The shopping mix is considered the most representative mix for this benchmark.

We use the Java servlets implementation from the University of Wisconsin [4]. The database scaling parameters are 10,000 items and 288,000 customers. This corresponds to a database size of 350MB.

For these experiments C-JDBC is configured without a cache but with parallel transactions and early response to updates and commits. The load balancing policy is Least Pending Requests First.

6.3. Browsing mix

Figure 10 shows the throughput in requests per minute as a function of the number of nodes using the browsing mix, for full and partial replication.

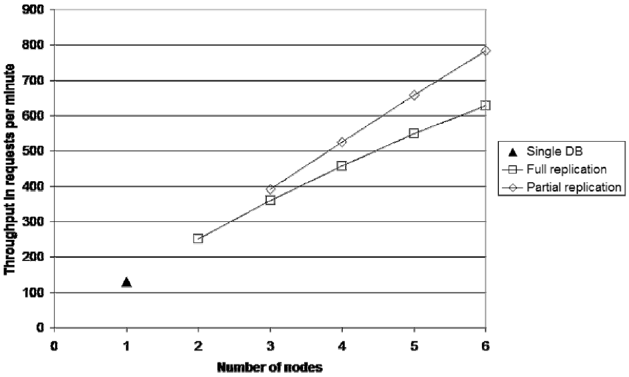


Figure 10. Maximum throughput in SQL requests per minute as a function of database backends using TPC-W browsing mix.

The single database configuration saturates at 129 requests per minute. Full replication starts with a throughput of 251 requests per minute with 2 nodes. The 6-node configuration reaches 628 requests per minute, representing a speedup of 4.9. This sub-linear speedup is due to the MySQL implementation of the best seller query. A temporary table needs to be created and dropped to perform this query. With full replication each backend does so, but only one backend performs the select on this table. Partial replication limits the temporary table creation to 2 backends. Partial replication improves full replication performance by 25% and achieves linear speedup. This example demonstrates the benefit of being able to specify partial replication on a per-table basis.

6.4. Shopping mix

Figure 11 reports the throughput in requests per minute as a function of the number of nodes for the shopping mix, which is considered the most representative workload.

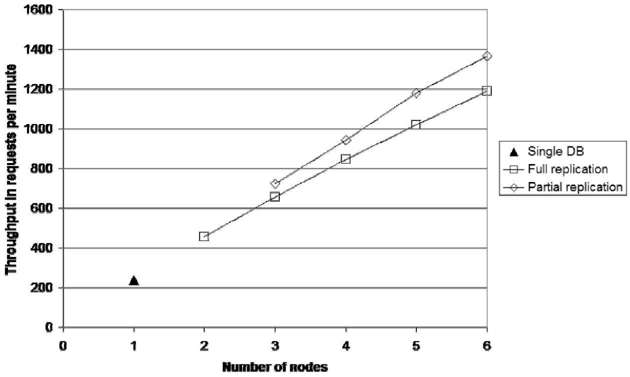


Figure 11. Maximum throughput in SQL requests per minute as a function of database backends using TPC-W shopping mix.

The single database without C-JDBC achieves 235 requests per minute at the peak point. Full replication achieves 1,188 requests per minute with 6 nodes. The shopping workload mix scales better than the browsing workload mix due to the smaller number of best seller queries. Partial replication again shows the benefits of per-table partial replication over full replication with a peak throughput of 1,367 requests per minute with 6 nodes.

6.5. Ordering mix

Figure 12 shows the results for the ordering mix for partial and full replication. The ordering features 50% read-only and 50% read/write interactions. Even in this scenario, with a large number of group communication messages as a result of the large fraction of updates in the workload, good scalability is achieved.

[1] These machines are old but they have a CPU vs I/O ratio comparable to recent workstations.

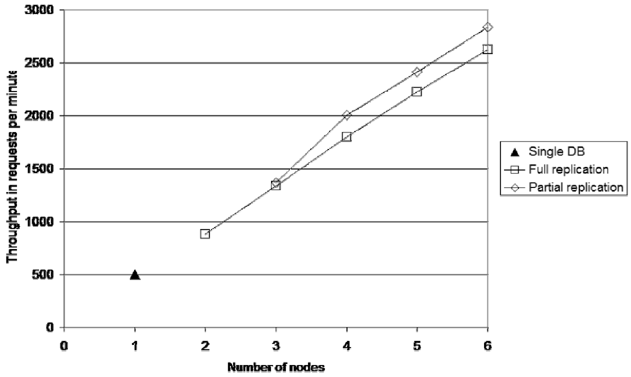


Figure 12. Maximum throughput in SQL requests per minute as a function of database backends using TPC-W ordering mix.

Full replication peaks at 2,623 requests per minute with 6 nodes, while partial replication achieves 2,839 requests per minute. The speedups over a single backend are 5.3 and 5.7 for full replication and partial replication, respectively.

6.6. Benefits of query result caching

It is also advantageous to use C-JDBC solely for its query result caching feature, even with only a single database backend. We have evaluated the benefits of query result caching using the servlet version of the RUBiS benchmark [1]. The RUBiS (Rice University Bidding System) benchmark models an auction site similar to eBay. We use the bidding mix workload that features 80% read-only interactions and 20% read-write interactions. Table 1 shows the results with and without C-JDBC query result caching.

RUBiS bidding mix *With 450 clients*	**No cache**	**Coherent cache**	**Relaxed cache**
Throughput (rq/min)	3892	4184	4215
Avg response time	801 ms	284 ms	134 ms
Database CPU load	100%	85%	20%
C-JDBC CPU load	-	15%	7%

Table 1. RUBiS benchmark (servlet version) performance improvement for a single MySQL backend with C-JDBC query result caching.

The peak throughput without caching is 3,892 requests per minute for 450 clients. The average response time perceived by the user is 801ms. The CPU load on the database is 100%. With C-JDBC and with consistent query result caching enabled, the peak throughput increases to 4,184 requests per minute, and the average response time is reduced by a factor of almost 3 to 284 ms. The CPU load on the database decreases to 85%,.

Further performance improvements can be obtained by relaxing the cache consistency. With a cache in which content can be out of date for up to 1 minute (entries are kept in the cache for 1 minute independent of any updates), peak throughput reaches 4,215 requests per minute, average response time drops to 134 ms, and CPU load on the database backend is reduced to 20%.

7. Conclusion

We presented Clustered JDBC (C-JDBC), a flexible and efficient middleware solution for database replication. By using the standard JDBC interface, C-JDBC works without modification with any application that uses JDBC and with any database engine (commercial or open-source) that provides a JDBC driver.

We have presented several use cases, illustrating how the C-JDBC's flexible configuration framework addresses user concerns such as high availability, heterogeneity support and performance scalability. Combining both horizontal and vertical scalability provide support for large-scale replicated databases. Query response caching improves performance further even in the case of a single database backend.

C-JDBC has been downloaded more than 15,000 times since its first beta release ten months ago. There is a growing community that shares its experience and provides support on the c-jdbc@objectweb.org mailing list. C-JDBC is an open-source project licensed under LGPL and is available for download from http://c-jdbc.objectweb.org.

8. Acknowledgements

We would like to thank Peter A. Daly of budget-ha.com for authorizing the reuse of materials from his web site. Lofi Dewanto from OpenUSS was very helpful and supportive of C-JDBC. J. Cameron Cooper provided us with a description of floodalert.org.

We are grateful to the C-JDBC user community for their feedback and support. We would like to thank all C-JDBC contributors who help us improve the software by testing it in various environments and contribute to the code base with fixes and new features.

Finally, we would like to thank our colleagues Anupam Chanda, Stephen Dropsho, Sameh Elnikety and Aravind Menon for their comments on earlier drafts of this paper.

9. References

[1] Cristiana Amza, Emmanuel Cecchet, Anupam Chanda, Alan Cox, Sameh Elnikety, Romer Gil, Julie Marguerite, Karthick Rajamani and Willy Zwaenepoel – Specification and Implementation of Dynamic Web Site Benchmarks – *IEEE 5th Annual Workshop on Workload Characterization (WWC-5)*, Austin, TX, USA, November 2002.

[2] Bela Ban – Design and Implementation of a Reliable Group Communication Toolkit for Java – Cornell University, September 1998.

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

[4] Todd Bezenek, Trey Cain, Ross Dickson, Timothy Heil, Milo Martin, Collin McCurdy, Ravi Rajwar, Eric Weglarz, Craig Zilles,

and Mikko Lipasti – Characterizing a Java Implementation of TPC-W – *3rd Workshop On Computer Architecture Evaluation Using Commercial Workloads* (CAECW), January 2000.

[5] Boris Bialek and Rav Ahuja – IBM DB2 Integrated Cluster Environment (ICE) for Linux – IBM Blueprint, May 2003.

[6] Budget-HA.com – High Availability Infrastructure on a budget – http://www.budget-ha.com.

[7] Caucho Technology – Resin 3.0 servlet container - http://www.caucho.com/resin-3.0/

[8] Emmanuel Cecchet, Anupam Chanda, Sameh Elnikety, Julie Marguerite and Willy Zwaenepoel – Performance Comparison of Middleware Architectures for Generating Dynamic Web Content - *Middleware 2003*, ACM/IFIP/USENIX International Middleware Conference, June 2003.

[9] Willy Chiu – Design for Scalability - an Update – *IBM High Volume Web Sites, Software Group*, technical article, april 2001. [9] Enhydra Java application server – http://enhydra.objectweb.org/.

[10] Enhydra Octopus – http://octopus.enhydra.org/.

[11] Jakarta Tomcat Servlet Engine – http://jakarta.apache.org/tomcat/.

[12] JMOB – Java Middleware Open Benchmarking – http://jmob.objectweb.org/.

[13] Bettina Kemme and Gustavo Alonso – Don’t be lazy, be consistent: Postgres-R, a new way to implement Database Replication – *Proceedings of the 26th International Conference on Very Large Databases*, September 2000.

[14] MySQL Reference Manual – MySQL AB, 2003.

[15] OpenUSS – Open Source Software for Universities and Faculties (Open Source University Support System) - http://openuss.sourceforge.net/.

[16] Oracle – Oracle9*i* Real Application Clusters – Oracle white paper, February 2002.

[17] D. Stacey – Replication: DB2, Oracle or Sybase – *Database Programming & Design 7*, 12.

[18] Transaction Processing Performance Council – http://www.tpc.org/.

[19] S. White, M. Fisher, R. Cattel, G. Hamilton and M. Hapner – *JDBC API Tutorial and Reference, Second Edition* – Addison-Wesley, ISBN 0-201-43328-1, november 2001.