Overview of HSQLDB Replication (HSQLDB/R)

Bela Ban Aug/Sept 2002
belaban@yahoo.com

This document describes the changes in the open source HSQLDB[1] codebase to provide database replication. The changes were relatively minor and localized. The goal for integration into the HSQLDB codebase is to avoid any overhead (CPU, memory) when running HSQLDB without replication. Ideally the replication code can be downloaded separately and replication can be enabled via a deployment configuration change.
HSQLDB version 1.7.2-alpha served as the basis for the modifications. JavaGroups 2.0.3 was used as a reliable transport between the nodes.
Database replication is when changes to one database are propagated (or replicated) to another database. As a example, we have 2 nodes (hosts) A and B, each running an HSQLDB server. Whenever a new table is created on A, the same table will also be created on B. Whenever data is inserted into the new table on A, the same rows will also appear in B‘s table.
Replication does not need to be unidirectional: in the above example it is possible for node B to create a new table, which will also be created in A, while at the same time node A inserts some data in an existing table. That data will also be inserted into the table on node B.
Also, replication is not restricted to 2 nodes: any number of nodes can be involved. For example, if we have nodes A, B and C, and node B decides to delete a row from a table, then that row will be deleted from all nodes.
Nodes are grouped by a name. All nodes in a system (e.g. a LAN) with the same group name (given as deployment parameter) are in the same replication group. Being in the same group means that all replicas of that group will receive each others updates. The management of the groups, plus the reliable dissemination of updates is done by JavaGroups.
When a new replica is started, and there is already an existing replica in the same group, the database state will be acquired from the existing member, the coordinator[2]. The new member then initializes itself from that state, and does not re-create its initial state from the database files. When there is no existing member yet, the newly started replica becomes the coordinator, and re-creates its database state from the database files (if existent).

1 JavaGroups

JavaGroups is the framework used to replicate database updates between the various HSQLDB servers. It ensures that all replicas receive the updates in a well-defined order (to be configured by the user). JavaGroups is also used to acquire the initial database state when a new replica starts, and to notify replicas when other replicas have crashed.

The main abstraction in JavaGroups is a **channel**, which is similar to a MulticastSocket; it is the handle to a group. A channel can be created and used to join a group. Once the group has been joined, a channel is used to send unicast and multicast messages and to receive messages. Finally a channel is closed, which removes the member from the current group.

Each channel has a **view**, which is an ordered list of member addresses (each member address consists of the member's IP address plus a port). Every member in a group has the same view, so in a group consisting of A, B and C (who joined the group in this order), the view in each member would be {A, B, C}, with A being the **coordinator**.

JavaGroups uses IP multicasting by default, which is very efficient in Local Area Networks, but it can be configured to use any transport. Configuration is a deployment issue, so replication could be configured to use a TCP-based transport rather than IP multicast to replicate HSQLDB servers across a Wide Area Network.

Some of the properties which JavaGroups provides, and which can be configured, include

[1] http://hsqldb.sourceforge.net
[2] The oldest member in a group is called the coordinator. When it dies, the second-oldest becomes the coordinator and so on.

- Reliable unicast and multicast messages. Dropped messages are retransmitted
- Fragmentation. Large messages are fragmented and reassembled at the receiver's side
- Ordering: FIFO (default), causal and total order
- Merge: after a network partition, members are joined back into one group
- Failure detection: crashed members are automatically removed from the view
- Group Membership: we know at all times who the current members in the group are, and get notified when a new member joins, or when a member leaves (voluntary removal) or crashes (involuntary removal)

The configuration of those properties can be done via an XML file, or via a simple configuration string. For HSQLDB/R either method can be selected (see details in chapter 4).

For more information on JavaGroups refer to http://www.javagroups.com.

2 Architecture of HSQLDB replication

The architecture of HSQLDB/R is shown below in Figure 1:

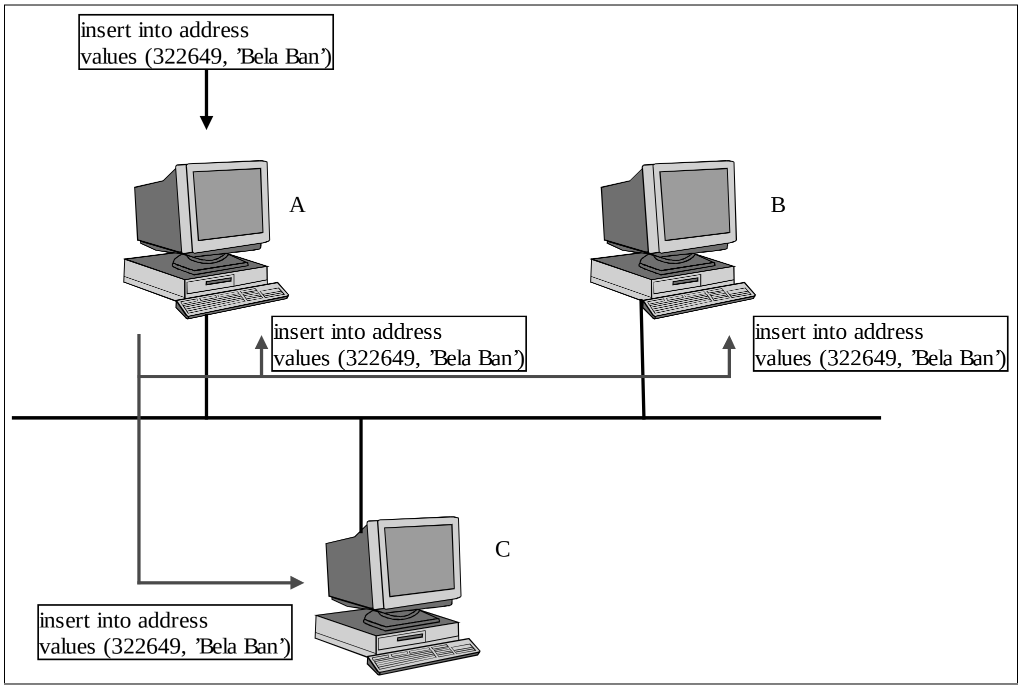
insert into address values (322649, 'Bela Ban')
A
B
insert into address values (322649, 'Bela Ban')
insert into address values (322649, 'Bela Ban')
C
insert into address values (322649, 'Bela Ban')

Figure 1

The HSQLDB servers A, B and C form a replication group. Note that A, B and C are processes, and can be located on the same machine[3], on different machines within the same Local Area Network, or even on different machines across a Wide Area Network. In this paper we discuss only the Local Area Network setup, where HSQLDB server would typically be started on different machines within the same network.
We assume that A was the first member started, did not discover any other members and therefore became the coordinator. In this case, it initialized its database from the *.data and *.script files.
The second member, B, discovered that A was already present, and therefore fetched its initial state from A. This was done by sending a message to A. A then dumped the contents of its database into a byte buffer which was sent back to B, which in turn used it to initialize its database. Note that B never even read its *.data and *.script files[4].

[3] In this case, the servers have to be started with different *.data and *.script files. Otherwise, they would overwrite each other's data.
[4] In this case, all files are renamed to a backup, before creating the new files. So, in case of data loss, the previous files can always be recovered.

When C is started it also acquires the state from A and renames its own local files so they are not overwritten by the freshly created database.
During operation, if there is an update in one of the databases, the update will be sent to all replicas. In the above example, a row is added to a table in A's database. Once the change is committed to A's database, it is propagated to all members of the group, in this case A, B and C. B and C will execute the statement against their own database, whereas A will reject it because it was the one who generated the update.

3 Installation and setup of HSQLDB/R

Note that the instructions below may change as we integrate the changes into HSQLDB/R. The ZIP file for HSQLDB/R contains both a copy of JavaGroups-2.0.3 and a modified version of HSQLDB-1.7.2-alpha. Once replication is integrated into the HSQLDB code base, we will remove the separate HSQLDB JAR file.
The following steps have to be taken to install HSQLDB/R:

1. Download hsqldbr.zip from http://www.javagroups.com (under the HSQLDB/R menu item) or from the download section under http://hsqldb.sourceforge.net.
2. Unzip the hsqldbr.zip file, e.g. into directory hsqldbr.
3. Include all the JAR files in your CLASSPATH[5]. If you are already using JavaGroups or HSQLDB, make sure to add the JARs at the head of your CLASSPATH.
4. Inclusion of the xerces*.jar files is optional. This is only needed if you want to use XML files to configure JavaGroups. However it is recommended because JavaGroups uses XML to configure its internal efficient marshalling mechanism. If XML support is not available, JavaGroups will fall back to a less efficient default marshalling mechanism. Note that any valid XML parser can be used here, it doesn't need to be Xerces.

3.1 System properties

Replication is enabled/disabled through system properties. The following properties are used:

- repl=<true | false>. If set to true, replication is enabled.
- group=<name>. The name of the replication group. All replicas that want to be in the same group have to used the same name. The default is "hsqldb-group".
- props=<properties>. The setup of the JavaGroups protocol stack. This can be a valid URL pointing to an XML file defining the JavaGroups properties, e.g. file:/home/bela/hsqldbr/state_transfer.xml. There is an XML file (state_transfer) included in the lib directory of the HSQLDB distribution. Note that, in order to use the XML format, a valid XML parser has to be in the CLASSPATH (see above). Alternatively, the old format for defining the JavaGroups properties can be used (see JavaGroups documentation for details). If no properties are given, the built-in default will be used.

The current convention is that, if one of these properties is set, replication is enabled, otherwise replication is disabled. So, by default, replication is off.
As an example, to start an HSQLDB server using replication with the default JavaGroups properties and a group name of "my-group", the following command line would be used:

```
java -Dgroup=my-group org.hsqldb.Server -port 5555 -database mydb
```

To connect a JDBC client to the database the following command can be used:

```
java org.hsqldb.util.DatabaseManager -url jdbc:hsqldb:hsql://localhost:5555
```

4 Running the demo

The following steps will show how to run 2 HSQLDB standalone servers and demonstrate how they replicate changes to the database to each other. The demo will run the 2 instances on the same machine, but in practice it will make more sense to use different machines.
Let's assume that the first instance will use database "primary" and the second instance will use "backup". By naming the databases differently, we can run both instances from the same directory, and they won't overwrite each other's files. We will use the XML method of defining the JavaGroups properties, and the

[5] Note that there are 2 scripts, run_primary.sh and run_backup.sh, which can be used to start 2 replicas. If the scripts are used, the CLASSPATH does not need to be set.

default group name of "hsqldb-group". (Note that you have to change the URL below to your own home directory).

To run the first instance execute

```
java -Drepl=true org.hsqldb.util.DatabaseManager -url jdbc:hsqldb:primary[6]
```

Alternatively you could use the following command line:

```
java -Dprops=file:/home/bela/hsqldbr/state_transfer.xml org.hsqldb.util.DatabaseManager
-url jdbc:hsqldb:primary
```

This would explicitly defined the properties for JavaGroups in the state_transfer.xml file. Note that the built-in properties are the same as the ones defined in state_transfer.xml, however using the latter one can experiment a bit with different properties (not recommended for the average user though).

This will run the first instance and create the files primary.properties, primary.data and primary.script in /home/bela. Now let's create a table and insert a number of values. Execute the following statements in the GUI:

```
CREATE TABLE TEST (ID INT, NAME VARCHAR(20));
INSERT INTO TEST VALUES (1, ’Bela Ban’);
INSERT INTO TEST VALUES (2, ’Fred Toussi’);
INSERT INTO TEST VALUES (3, ’Marc Fleury’);
```

Now start the second instance by executing the following command:

```
java -repl=true org.hsqldb.util.DatabaseManager -url jdbc:hsqldb:backup[7]
```

A second instance will be started. Since it finds an already running instance, it will fetch the database contents from the primary. Its tree on the left side should show the previously created TEST database. Now do a

```
SELECT * FROM TEST
```

and you will see the 3 rows that are presently in the TEST table.

Now go to the backup server and add an additional row:

```
INSERT INTO TEST VALUES(4, ’Bill Gates’);
```

Now go to the primary and execute

```
SELECT * FROM TEST
```

You'll see that the results are

```
ID NAME
-- -----------
1  Bela Ban
2  Fred Toussi
3  Marc Fleury
4  Bill Gates

4 row(s) in 1 ms
```

This means that the new row was successfully replicated from the backup to the primary.

Now go to the primary and execute

[6] On UNIX systems, there is a script (run_primary.sh), which does this for you.
[7] Alternatively, ./run_backup.sh could be used on UNIX systems.

```
DELETE FROM TEST WHERE NAME LIKE ’%Gates%’;
```

You will see that the row is deleted from both the primary and backup servers.

Any SQL statement which updates the database will be replicated to all the replicas. Note that - when creating or dropping tables - the "refresh tree" menu item needs to be activated so that the changes are visible in the GUI.

Now exit the primary GUI and restart it. You will notice that the database contents are not read from the database files, but from the backup server instead. Also, the primary.* database files have been renamed to primary*.backup, and new primary.* files have been created.

5 Use cases for startup

This section describes some typical use cases related to how new replicas can acquire the database state, and how changes to a database are replicated.

5.1 First node (primary)

When a new replica is started, it first checks whether there are already existing members in the replica’s group, from whom to acquire the initial database state. This case assumes that there are none, so the database will be initialized from the database files (if there are any). Since the replica is the first member of the replica group, it will serve as coordinator. All future members will fetch their initial database state from this member. If the member dies, the second oldest member will take over as coordinator.

5.2 Second node (backup) with state transfer

In this case, there are already members in the replica group, therefore the initial database state will be fetched from the current coordinator (first and oldest member of the group). The new replica sends a message to the coordinator to ask for the current group state. The coordinator then returns a byte buffer as response, which is then used by the new member to initialize its database. Note that the new member does not read its database contents from any files. Instead, existing files will be backed up and new ones created, to which the acquired state will be written.
Note that acquiring the state from the database may require a short read-only access to the coordinator’s database (currently not implemented), during which the contents of the coordinator’s database are dumped into a byte buffer to be sent back to the new member.
While the new member intitializes its state from the byte buffer received from he coordinator, it may be possible that it already receives some updates to the database via the replication. JavaGroups will simply discard them and retransmit them at a later stage when the new member is fully initialized.
The JavaGroups state transfer protocol ensures that spurious replication messages received before the state was dumped, will be discarded (as they are already part of the state), and later messages will be correctly received (or retransmitted), and subsequently applied to the state. This means that a new member can join a replica group and acquire the state while replication messages are exchanged between the group members. Thus replication does not have to stop while a new member is admitted. The only period where the coordinator’s database is inaccessible is the time during which it dumps its current state[8].

5.2.1 Transfer of large state

When the state to be transmitted is large, it may be better to create a snapshot of the coordinator’s database and have the coordinator start writing subsequent transactions to a log, and then initializing a new replica from the snapshot and subsequently only fetching the contents of the log from the coordinator. This is currently not implemented.
Note: we need to test large states with the current state transfer (blocking of coordinator, fragmentation etc).

5.3 Update propagation through JavaGroups

When a transaction is HSQLDB is about to be committed, there are several ways to replicate it to all replicas. They are described below.

[8] Note that read-only functionality is not implemented in the current prototype.

5.3.1 Commit transaction locally, then propagate (discard own multicast)

This is the currently implemented solution. A transaction is committed to the local database first. Then we check whether the transaction modified the database and - if yes - send a multicast with the SQL statement to the replication group. For example, a SELECT statement would not trigger a multicast, whereas an INSERT or UPDATE statement would.
The message sent to all members contains the SQL statement and the sender of the message. Since the replication message is sent to all members, the sender will receive it too. The sender will discard its own messages, otherwise it would generate errors.
This method of update propagation has the advantage that it is very efficient (caller never has to wait) and simple to implement. The disadvantage is that, when several replicas update the same data, there will be inconsistencies. For example, when two replicas want to insert a row with the same primary key, both local transactions will succeeed, but when the insertions are replicated, they will be rejected because the rows already exist. See the more costly solution below on how to solve this problem.
This solution is feasible in the following cases:

- Only one replica (e.g. the primary) is used for updates. Other replicas (backup(s)) are only used to receive the replicated updates and to serve as new primary in case of failover.
- Replicas maintain disjunct datasets; for example the primary is the main access point for rows 1 and 4, whereas the backup is the main access point for rows 2 and 3. Rows 1 and 4 are replicated to the backup, whereas 2 and 3 are replicated to the primary. This is often the case with servlets/EJBs, where the creation of 'sticky sessions' is load balanced across primary and backup, but - once a session has been created in a web/app server - the same server is used for the lifetime of the session, or until the server dies.

5.3.2 Propagate update first, update self when multicast received

Note that this alternative is currently not implemented. Here, we don't commit the transaction, but instead generate a multicast, which is sent to all members. On reception, every replica applies the transaction to its local database. The advantage is that JavaGroups can control the order in which transactions are received by all the members of a replica group. The (small) disadvantage is that the committer of a transaction has to block until it received its copy of the multicast replication message it generated. This is generally not a problem since local loopback at the IP layer should result in an almost immediate reception of our own multicast.
As will be shown later (section 6.3.2), having JavaGroups control the order in which transactions are applied can be important to avoid update conflicts, e.g. two replicas inserting the same row (identical primary key) into the same table.

5.3.3 Immediate propagation versus periodic propagation

In the case where transactions are committed to the database immediately and only then multicast to the other members, it may be useful to bundle multiple transactions into one, and to send them at scheduled intervals.
For example, when a primary database is used all the time for access, and the backup serves only to take over in case the primary fails, then it might be feasible to schedule replication to take place every 5 minutes. In this case, when the primary fails, we might have up to 5 minutes worth of lost data. When such a loss can be tolerated, then this solution might be feasible.
Another example is a company's central personnel database: changes are infrequent and data can be reconstructed from paper. If the company has subsidiaries across the country, then the central database may be replicated to each of the subsidiaries via dial-up connections. In such a case it may be most cost-efficient to replicate a day's worth of data with a cron job every weekday at midnight. All the updates will be queued locally (e.g. in another database table), and then sent in one bundle to all subsidiaries.

6 Replica access patterns

This section discusses the 3 most anticipated access patterns for HSQLDB replication: access only to the primary, access to primary and backup, with disjunct datasets, and access to the same database in both primary and backup.

6.1 Access only to primary, failover to backup

In this case, all access is only to the primary replica. Read-only access may be to any replica, although the data may be dirty (e.g. updated in the primary, but not yet replicated). Since access is only to the primary, there will never be any conflict caused by replication.

This case is suitable for use when the backup database serves as a warm standby, and can take over in case the primary fails. Data is replicated only from primary to backup.

6.2 Access to primary and backup, different data (no conflicts)

In this case, both primary and backup will be accessed, but there will not be any conflicts caused by simultaneous access to the same data because the data is partitioned such that they don't overlap. As shown in the example in Figure 2, the primary hosts datasets A and C, and the backup hosts datasets B and D, and {A,C} and {B,D} are disjoint. All access to dataset A and C goes to the primary, which replicates A and C to the backup. By the same token, all access to B and D goes to the backup, which replicates B and D to the primary.
In case the primary fails, datasets A and C will be accessed through the backup, which now serves as the main access point for {A,C} *and* {B,D}. When the primary comes back up (as backup) it will serve merely as replication destination for {A,B,C,D}, which are now hosted on the backup (new primary).
The logic for how to access datasets is outside the scope of this article, but is common practice for example in the web server area (sticky sessions).
This example shows that careful partitioning of the data allows for simultaneous access to data replicated over multiple databases without access conflicts. Of course, the application needs to be written in such a way that it enforces that access to certain datasets are always directed to the same database server.

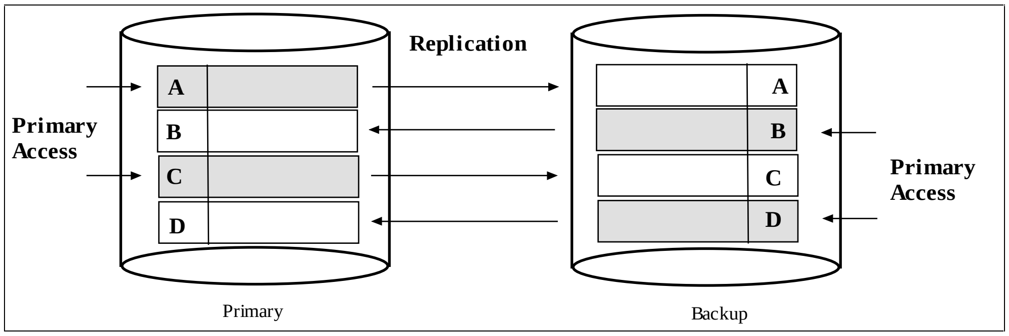

Figure 2

6.3 Access to primary and backup, same data (conflicts)

This case allows for simultaneous access to the same data (e.g. same primary keys) on different replicas, which can lead to replication conflicts if the currently implemented replication mechanism (described in section 5.3.1) is used.
This section discusses two mechanism for allowing for simultaneous data access, without generating conflicts. Note that these mechanisms are not currently implemented in HSQLDB/R. The reason for discussing them is that one of them will eventually be chosen and implemented, based on input from the HSQLDB community.

6.3.1 Conflict resolution using globally unique IDs

The primary keys are usually the sorts of constraints that cause data conflicts. For example, if a customer with an ID of 322639 is already in a table which has a primary key on customer IDs, then attempting to insert a customer with the same ID will cause a conflict.
Primary keys are usually unique per database. However, if an application can use primary keys that are globally unique, then there won't be any conflicts. For example, if it is possible to add the hostname to the primary key, and all updates carry the hostname, then we can prevent conflicts. In the above case, there could be two simultaneous updates; one with with a customer ID of A:322649, and the other on host B with a customer ID of B:322649. This will prevent conflicts, but probably defies the purpose: here the customers A:322649 and B:322649 were probably meant to be the same customer !

(* This is complete baloney, either remove or look for a better example *)

6.3.2 Conflict resolution using total order

By establishing a total order over all updates that are replicated to the group members, we can ensure that every member will see the exact same sequence of transactions. This is done by using the total order protocols of JavaGroups.

Let’s assume that a new customer with ID=322649 (ID is the primary key) is created in replica A and almost simultaneously in B. Let’s call the transaction on A T1 and the one on B T2. Let’s further assume that the data associated with that customer is not exactly the same, e.g. T1 might have as phone number the customer’s work phone, whereas T2 has no phone at all.
With the simple mechanism described in section 5.3.1, A will apply T1 (the one with the phone number) and then replicate the update. B will apply T2 (the one without the phone number) and then trigger the replication. When B receives the T1 replicated from A, it will cause a conflict because a customer with ID=322649 is already in the database. Therefore, the replicated T1 will be discarded at B. Vice versa, A will discard the replicated T2 because of the duplicate primary key. Therefore we end up with both replicas having a customer with ID=322649, but slightly different data associated with it. Depending on which replica is accessed, we will receive different data.
With the mechanism described in 5.3.2, but without total order, both T1 and T2 are multicast first, before being received by the respective group members. Since there is no total order involved (in the default JavaGroups configuration), A might receive T1 and T2, and B might receive T2 and T1. In this case we will also incur differences between the data associated with the same customer.
If we introduce total order, both A and B will either receive T1 followed by T2, or T2 followed by T1, but they will not receive T1 followed by T2 at one site and T2 followed by T1 at the other site (or vice versa). This means that the transaction that happens to be first, will create the customer with ID=322649, and the second transaction will be discarded. This is perfectly okay, as the creator of the second (failed) transaction will get an exception when trying to commit that transaction. Total order ensures that the data associated with the same customer is exactly the same.
In the case of updates (e.g. the UPDATE or SET statements), the same mechanism can be used: when two updates U1 and U2 to the same row are multicast, JavaGroups ensure that every replica either receives U1 followed by U2, or U2 followed by U1. Because the second transaction does not cause any data conflicts (as with primary key insertion), the second update will overwrite the first update’s changes.
Note that total order is a heavy-duty protocol: every multicast is first sent to a coordinator who then multicasts on behalf of that member (sequencer-based total order protocol), or group members constantly circulate a token around a virtual ring imposed on the group (total token based ordering).

6.3.3 Conflict resolution using distributed locks

This solution is slightly less costly, and can do without the total order protocols. However, with many updates, this approach may generate to much traffic on the network.
The trick used to ensure that updates that are consistent across a cluster of replicas is to lock the replicas before sending out a multicast, and to unlock them again when done. Locks are lease-based synchronization points with a timeout[9]; they expire automatically after a given amount of time. This prevents a single sender from monopolizing a lock for too long. Also, it prevents stale locks in case of crashed replicas. When a member crashes, all locks held by that member will automatically be removed.
Let’s look at the example of creating a new row in the customer table with customer ID=322649. Let’s assume both A and B create this new row simultaneously. A now sends out a multicast attempting to lock the customer table in both A and B. At the same time, B does the same. A was slightly faster and manages to lock the table in both replicas. At this point it can go ahead and send another multicast with the update, namely the insertion of the new row. Both insertions will succeed. Now B attempts to lock the customer table in A and B. In both cases, since the lock has already been acquired, B has to wait. Now A unlocks the two tables, which allows for B to proceed. However, since there is already a row with the same primary key, both of B’s updates will fail. In this case, B notifies the creator of the transaction of the failure (e.g. by throwing an exception) and unlocks the rows. (Note that the sending of the update plus the unlock operation can probably be combined into a single message.)
A lock has a timeout associated with it (typically a number of seconds), which will prevent a sender from monopolizing a resource, and also prevent deadlocks. Consider the case where A manages to lock A’s customer table, and waits for B’s locks to be released, while at the same time, B waits for A’s lock. This would result in a deadlock, unless one or both of the locks would be released by the timeout. In this case, both A and B will retry acquiring the locks, but only after a random timeout, which minimizes the chances that they will try to acquire the same locks at exactly the same time.
Locks would have to be attached to various resources: e.g. the creation of a new table would need a lock on the database itself, while the insertion or update of a row would only need to lock a specific table. Note that, if we can determine which row(s) need to be locked, we could lock individual rows rather than the entire table, making this solution much more fine-grained, thus allowing more concurrent transactions.

[9] A timeout of 0 means to wait forever.

7 Changes to HSQLDB

The main changes affect Database and Log. Also there is a new class ReplicationData, plus some JARs and a configuration file. With the exception of the javagroups.jar file, all JAR files and the configuration file are optional, depending on how JavaGroups is run.

7.1 org.hsqldb.Log

2 methods were added to Log in order to initialize the Log from state sent to us over the network (state retrieved from an existing replica), and to dump the current database into a byte buffer:

scriptToBuffer() is a copy of scriptToFile(), but dumping the DB contents into a file they are dumped into a byte buffer. Both methods should probably be merged, so that scriptToBuffer() generates the buffer and scriptToFile() writes that buffer to a file

initializeFromBuffer(): this method takes a buffer (received from the coordinator) and initializes itself from it

Maybe Log should provide methods byte[] toBuffer() and fromBuffer(byte[]), and other methods, such as scriptToBuffer(), should internally make use of them

7.2 org.hsqldb.Database

The constructor of Database creates a new JavaGroups channel (if replication is enabled), fetches the state from the coordinator (if available) and initializes its database from it. In addition, it registers a listener which listens for replication multicasts and applies them to its database (unless sent from self, in which case it will be discarded).
The execute() method now checks which statement modifies the database, and multicasts the update after is has been committed.
The getState() and setState() methods are called by JavaGroups to fetch or set the database state respectively. In the first case te logger dumps the database contents into a byte buffer, whereas in the latter case the byte buffer received as argument is used to construct the contents of the database.

7.3 org.hsqldb.replication.ReplicationData

Can be moved to one of the existing package. Contains the data (SQL statement) which is sent via replication multicasts.

7.4 Additional JAR files

These are
- javagroups-core.jar
- xercesimpl-2.1.0.jar
- xercesxmlapi-2.1.0.jar

The first one contains JavaGroups and needs to be included in the CLASSPATH if replication is used. The Xerces JAR files are optional, and are only required when XML files are used to configure the JavaGroups properties. However, JavaGroups uses XML internally, so we suggest to always include these two files in the CLASSPATH.

7.5 Configuration file

The state_transfer.xml file defines the JavaGroups properties needed for replication (e.g. it includes the state transfer protocol used to fetch the database state from the coordinator).

8 Proposal for integration of replication into HSQLDB

8.1 Database and Log classes determined by Factory

The creation of these classes should be governed by factories. Instead of

```
db=new Database();
```

we should say

```
db=DatabaseFactory.createDatabase();
```

This would help creating subclasses of Database and Log (and others), e.g. ReplicationDatabase. DatabaseFactory would be configured to create ReplicationDatabases (subclass of Database) rather than Databases in case replication is enabled. This would enabled to integrate replication in a non-intrusive manner. By the same token, if replication is disabled, the application would not have to pay for any overhead that it doesn't need (e.g. additional classes loaded etc).

8.2 Database constructor

Make the Database constructor small and call init() or a number of initialization methods. This way, subclasses can selectively override which initialization method whould be overridden, instead of having to override the entire constructor.

9 Issues

9.1 Failover-aware JDBC driver on client side

JDBC driver should have a means of knowing which DB servers are in its group. It should always access the same database server, but when that server fails, it should automatically fail over to another database server. JavaGroups Services could be used for the implementation of this feature.

9.2 Quiescing a database during content dump (read-only state)

Nice to have. Feature needs to be provided by HSQLDB.

9.3 Partitioning and merging

What happens when a partition heals ? Apps will get a viewChange() callback with a MergeView instead of a View. How do we merge databases ?

9.4 Binary state transfer and replication

Getting the state as a list of SQL statements is slow, because each SQL statement has to be parsed in order to be applied to the database. Same goes for replication. We need an efficient internal binary representation of the database contents (table schema, table contents, users, stored procedures etc). This will speed up both state transfer and replication itself.

9.5 Notifications when underlying database/table was changed because of replication

It would be nice to offer an interface for replication-aware applications that provides callbacks when the underlying database has been modified, e.g. by data being replicated into its database. A Gui for instance might change its view based on the updated data.

9.6 Replication across Wide Area Networks

JavaGroups can be configured to operate across WANs: this is simply a deployment issue. One scenario is to have the primary HSQLDB server in one location, and the backup in a geographically different location. Depending on the bandwith available between the server, the data would be replicated to the backup immediately, or periodically.
In case of a catastrophic event, most of the data that was available on the primary (depending on the replication frequency) would be available on the backup, and applications could be switched over to use the backup.