Global Peer-to-Peer Data Replication Using Sybase Replication Server

By Mich Talebzadeh

Mich Talebzadeh is an independent Sybase consultant, working largely in investment banking. He teaches on topics including SQL Server administration, performance and tuning, data replication, and client/server database and application design. He can be reached at: mitch.talebzadeh@db.com

I recently worked on the implementation of a global peer-to-peer data replication for a Front Office Trading System (FOTS) utilizing Sybase replication. This work was carried out as part of a major requirement for a global investment bank. The FOTS provides traders with a view of the trading activity and the positions held, allowing them to continue trading against orders placed earlier by other offices in other locations. For example, a trade may include the exchange of securities in multiple geographic regions such as London, New York, Tokyo, and Hong Kong.

The work was actually started by building a Sybase Warm Standby for the London database. Having successfully implemented this utility, a pilot project was put in place to check the feasibility of one-way replication to the Hong Kong Data Server, effectively testing the volume of data replicated and stress-testing the WAN. As the application relied on a third-party package, care was taken to avoid changing the code. The data structure was enhanced by addition of primary keys (prerequisites for warm standby replication). In peer-to-peer replication, a given database acts as publisher and subscriber simultaneously; all databases play equally important roles.

Having identified the initial problems, a truly peer-to-peer replication system was set up among London, Hong Kong and Tokyo, effectively replicating almost all the tables. The information on all sites had to be as current as possible and had to be available 24 hours a day. Practically in excess of 95% of the transactions had to be applied to all databases, worldwide, within five minutes of data being posted to a local database.

This article provides templates of how to create replication and subscription definitions for user tables in a given Sybase database, as well as practical examples of how to apply function strings to tailor what is delivered to the destination table. In addition, we'll discuss ways of monitoring the replication system and quickly checking the data.

Project Problems to be Addressed

A typical trading system needs to provide facilities to a business community scattered around geographically distant sites; in our case, London, Tokyo, Hong Kong and New York. A trader logs into the Application Server (a UNIX server) locally through an X-Windows session and starts the application, which connects to the Data Server via one or more threads. In the majority of cases this set-up involves many Application Servers running locally, with the Data Server located in one of the major sites. In our example, the Data Server was located in London. However, this set-up has the following drawbacks:

- Application performance is limited because of the geographical distance.
- Network response degrades when traffic over the WAN is heavy. For exam-

ple, when users in remote sites run reports requesting a large number of rows, there is an impact on those entering trades etc.

- The Data Server can potentially become a bottleneck as a larger user community contends for access.
- Data becomes unavailable when a failure occurs on the network.
- There are side effects on database maintenance. For example, Update Statistics, Database Consistency Checks (DBCC), and other tasks are performed when remote users are busy putting in trades. This introduces unnecessary complications and delays.
- A typical third-party application may not access data in the most efficient way. For example, if the application makes a large number of discreet queries to the database, connection latency between the Application and Data Server could cause start-up delays.
- If the application were to crash, the system becomes unusable to traders and has an unacceptable business impact.

We responded to these issues with a three-step plan. First, we created the Sybase Warm Standby database for the local site on the standby (BCP) server. Next, we created a peer-to-peer replication system for the trading database with local copies of the database in London, Tokyo and Hong Kong respectively. The database in each site acts as the source and recipient of data—in other words, the database plays the role of publisher and subscriber simultaneously. Third, we created a set of utilities to monitor and maintain the replication system. In such a trading system, service availability and recovery is essential, so particular attention was given to early warning systems.

Where to Locate the Replication Server

The Sybase Replication Server is an open server and could be located on any UNIX Server. It is advisable *not* to put the Replication Server on the production Data Server. This keeps the production box simple and less complicated (a plus for maintenance and recovery), allows us to resource the CPUs and memory for the production SQL Server, and keeps the Replication Server up and running even when the production box is down. The Replication Server was created on the BCP server, which served largely as an emergency backup.

The Warm Standby Setup

I'd like to say a few words on our warm standby set-up (valid for v. 11.0.3). A warm standby set-up is a pair of databases, one of which is a backup copy of the other. Client applications update the active database; Replication Server maintains the standby database. If the active database fails, switching to the standby database allows client applications to resume work with little interruption. A warm standby database is basically a simplified form of one way replication.

Sybase Warm Standby replication will only replicate the data to the standby box, and the standby database can only be used in read only mode. Of course, there can only be one warm standby set-up for a given database. Any changes to the objects in the database (i.e., database related application patches) will not be replicated and will have to be applied to the production and standby databases. Login names, database users, and permissions are not replicated. Although Replication Server does not usually require replication definitions in order to maintain a standby database, it does use any replication definitions that exist. Note that you need to generate replication definitions for tables with more than 128 columns.

We also created primary keys for replicated tables. In a warm standby set-up, the Replication Server generates a **where** clause to specify target rows for updates and deletes. If there is no replication definition for a table, the generated clause includes all columns in the table, except text, image, timestamp, and sensitivity columns as the primary key. This turns out to be inefficient.

Also, don't forget that you still need to prune the transaction log of the primary database on a regular basis. Just one option would be to dump the production database daily at 6:00 a.m., followed by dumping of the transaction log at regular intervals between 7:00 a.m. until 9:00 p.m. After 9:00 p.m., turn on the **truncate log on chkpt** option on the production database. You need to fit this to your schedule.

Planning for the Peer-to-Peer Replication System

In planning a global replication system, I suggest that administrators take the following ideas under consideration. First and foremost, of course, some business rules have to be defined in conjunction with the business in order to guarantee the integrity of data on each site.

Then, when beginning to test your system, set up a one-way replication between two locations, identifying the primary site (in our case, London) and the subscriber site (Hong Kong). Of course, the best option is to set up the test environment on one local and one remote server. Otherwise, you can use two different servers locally linked via a WAN simulator. Note that a one-way set-up can be extended to a peer-to-peer replication.

It is important to estimate the volume of data to be replicated daily, as well as the daily growth of your database. Also, you should establish the bandwidth between the two replicate sites, letting you know whether you should embark on replication or if the bandwidth needs to be improved. In addition, you must establish how the replication is going to be achieved. Do you need to:

- Replicate tables and turn off triggers for replicated transactions?
- Not replicate tables and let triggers insert the records?
- Replicate stored procedures?

Does your application depend on timestamps for certain transactions? Remote locations mean different time zones—a data insert at 9:00 a.m. in London corresponds to a replicated transaction of 9:00 a.m. in Hong Kong (this assumes no latency), whereas the local time would be 5:00 p.m. Would this be considered a valid business transaction? If not, consideration should be given to the use of function strings to allow for the local timestamps.

Using Table Replication

On our project, it was decided after testing to replicate tables and turn off triggers for replicated transactions. With this solution we need to identify those tables to be replicated and establish whether they meet the criteria—are primary keys defined, etc.—and we need to exclude local or static tables, which can be loaded at the start-up when the primary database is loaded onto the remote sites. We exclude the identity column from replication, as the remote server will automatically generate identity columns for replicated transactions.

Managing Conflicts in Peer-to-Peer Replication

Managing Inserts

A typical local table will include inserts from local applications in addition to inserts delivered via replicated transactions. In a peer-to-peer set-up, both tables are bi-directional. Designers tend to use unique IDs to identify records in a table, and primary keys or unique clustered indexes are usually built on the unique ID. The unique ID for a given table tends to be a monolithically incrementing 32-bit integer stored in and retrieved from a table, the so-called *table_next_row_id*. Intersite conflicts occur when rows are inserted in a local table and distributed to the remote table. If the remote table already has a record with the same unique ID, the replicated insert will be rejected, and the data at the remote table will be inconsistent with the local table. Possible solutions are:

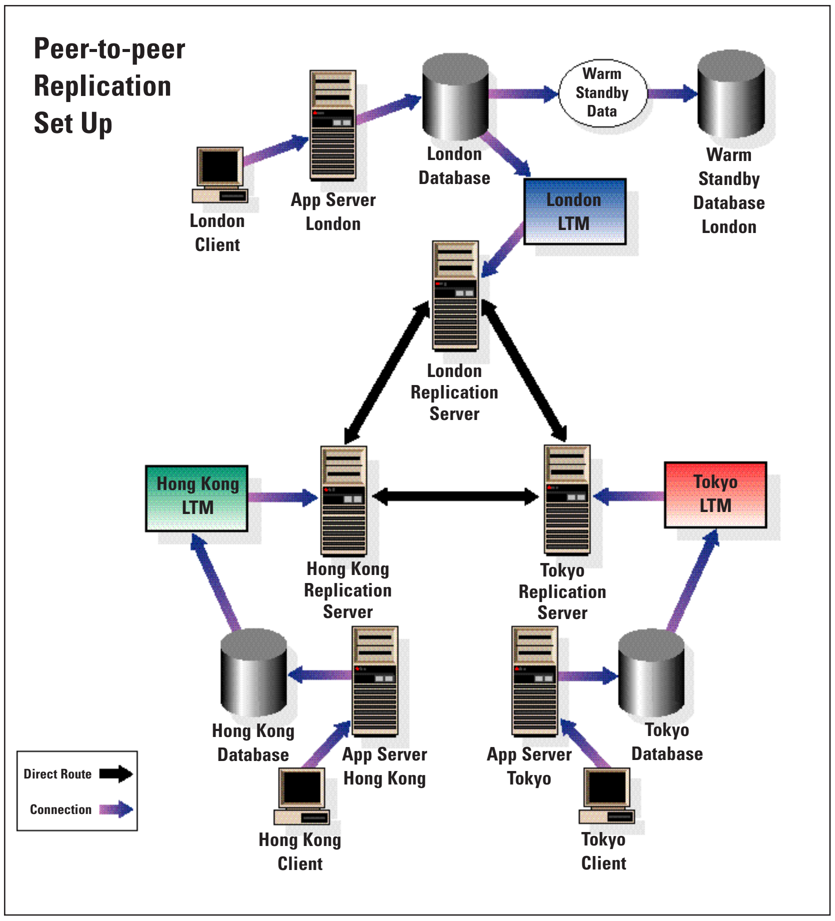
1. Add a location key to the tables if it is not already there, and include it in the primary key for replicated tables. This is useful if the application is at the design stage. As replication implementation is normally an afterthought, this approach may not be possible without a substantial change to database schema and stored procedures (establishing ownership of data).
2. Localize the *table_next_row_id* and do not replicate it. Allocate ranges for the *next_row_id* column for each location (a 32-bit integer provides ability to store up to 2 billion unique values). For example, you can allocate the following ranges:

Location	Reserved range for next_row_id Column
London	1-100,000,000
Hong Kong	101,000,000-150,000,000
Tokyo	151,000,000-200,000,000

Conflicting Updates

The best way to handle conflicting updates from different sites is to construct the application and the environment so that conflicts cannot happen. However, in many cases one needs to rely on application-specific business rules to reduce/eliminate the conflicts. For a trading system these could be:

- On *performing a simultaneous new trade* on the same holding on a different site, the problem will occur on a calculated field, such as quantity or P&L. Furthermore, there are no signals to warn the users when the problem occurs. The adopted solution is to recalculate these fields nightly so that they will have the correct figure by the following day, when the portfolio is loaded. However, we have not yet encountered such a problem.
- On *simultaneous update on the same order*. This could happen due to a mistake. The business rule is whoever created the order owns it, and should be the one who updates the order. If this happens, the quantity (P&L) data will be out of sync. Again, there will be no warning message to indicate this and it will be very difficult for IT support to detect it. The traders will inform Application Support that the P&L or quantity is wrong. The support group needs to check the historical order and amend it appropriately. Once this is carried out, the correct details will be replicated to other sites and the databases will be in sync again.



Building three replication servers using sybase rs_init facility

Peer-to-Peer Replication Implementation

In our set-up, we are replicating data between London, Hong Kong, and Tokyo. The majority of the tasks mentioned below are best performed when no user is using the databases, such as over weekends. Each database is controlled by its local replication server.

Replication Server	Location	Location of RSSD	ID server
lon_rep_server	London, on Standby Server	Standby sql server	yes
hk_rep_server	Hong Kong	Hong Kong sql server	-
tyo_rep_server	Tokyo	Tokyo sql server	-

1. Before creating replication servers ensure that you have already created devices for the RSSD databases and have raw partition devices for the stable queues, etc. In other words, fill in those Replication worksheets.
2. Once the Replication Server is created, change its password using the **alter user** command from the Replication Server.
3. Create the diagnostic run files for replication servers so the DBA may observe each replicated transaction performed by the server (invaluable in identifying problems). This is achieved by replacing repserver binary with repserver.diag binary in the run file and adding the following entry to the replication server .cfg file:

```
trace=DSI,DSI_CMD_DUMP
```

4. Create error class rs_sqlserver_error_class (default sql server error class) in ID replication server only. This will handle sql server errors in replication server. The default error action for all errors returned by sql server is to stop replication. You can assign action to the created error class etc. Error actions are stored in table rs_erroractions. Use rs_helperror error_no, v to get information about the errors.
5. Turn off trigger settings for London, Hong Kong and Tokyo servers in the corresponding replication server. Use configure connection command with dsi keep triggers option set to "off." For example, in lon_rep_server run the following command:

```
configure connection to london_sql_server.db
set dsi_keep_triggers to 'off'
```

6. Add the server name and port ID of replication servers to the relevant interfaces files.

Creating Direct Routes

In our triangle diagram, we need to create routes in order for our three replication servers to send messages to destination replication servers. A route is a one-way message stream that sends requests from one Replication Server to another, carrying data modification commands, replicated functions, and stored procedures. In this design the routes are created as follows: For example in lon_rep_server run the following command to create route to hk_rep_server:

Route type	Source	destination
Direct	lon_rep_server	hk_rep_server
Direct	lon_rep_server	tyo_rep_server
Direct	hk_rep_server	lon_rep_server
Direct	hk_rep_server	tyo_rep_server
Direct	tyo_rep_server	lon_rep_server
Direct	tyo_rep_server	hk_rep_server

```
create route to hk_rep_server
set username hk_rep_server_rsi
set password hk_rep_server_rsi_ps
```

hk_rep_server_rsi is the RSI username already created by rs_init when you created the hk_rep_server. *hk_rep_server_rsi_ps* is the default password for such user etc. Use *rs_helproute* in any RSSD to check the status of the route created.

Loading the Database to Be Replicated

In order to perform the initial load of the database to be replicated, you should perform the following steps:

1. Decide where you are going to load your initial database. In our case, we chose London.
2. dbcc the database and perform update statistics in London.
3. Review all the primary keys for tables to be replicated.
4. Turn off all replication flags in the user tables using *sp_setreptable table_name, false*.
5. Do **dbcc settrunc(ltm,ignore)** on the database.
6. Dump transaction with truncate_only.
7. Dump database to the dump directory.
8. FTP the dump file to the warm standby server.
9. Load the database on the standby.
10. Zip the dump file to remote locations.
11. Load the database from the dump file in remote locations.
12. Localize the so-called local tables. For example, if you have *table_next_row_id, set next_row_id* column to the appropriate starting values for location.

Adding databases is quite straightforward: To add to the London Production database, we use *lon_rep_server*; for Hong Kong, use *hk_rep_server*; and for Tokyo, use *tyo_rep_server*. Note that all three databases are a source of data and therefore require an LTM or a rep agent.

Creating Replication Definitions

Once the databases are loaded, we can create all the replication definitions for the London database tables using lon_rep_server. You need to pass the *sql server name* and the *database name* as parameters. For a script (genrepdef.ksh)that will automatically create replication definitions, see the extended version of this article on the ISUG website at www.isug.com. Also, check that replication definitions are successfully implemented by looking at the log files and using rs_helprep in the relevant RSSD database.

To create replication definitions in the other publisher sites, this would be:

Replication Definition	Sql server named passed to script	Database name passed to script	Replication server run against
London tables	london_sql_server	db	lon_rep_server
Hong Kong tables	hk_sql_server	db	hk_rep_server
Tokyo tables	tokyo_sql server	db	tyo_rep_server

Next, turn on replication for all replicated tables by running *sp_setreptable table_name, true*. If you use the command *rs_helprep* in any RSSD database, you should see all replication definitions for all sites irrespective of which RSSD you are looking at. Finally, create subscription definitions for all other sites by running the above script (*genrepdef.ksh)* against the local replication server.

Creating Subscription Definitions

You should create two subscription definitions for each replication definition:

Subscription Definition	Sql server named passed to script	Database name passed to script	Replication server run against
London to Hong Kong	hk sql server	db	hk_rep_server
London to Tokyo	Tokyo sql server	db	tyo_rep_server
Hong Kong to London	london sql server	db	lon_rep_server
Hong Kong to Tokyo	tokyo sql server	db	tyo_rep_server
Tokyo to London	london sql server	db	lon_rep_server
Tokyo to Hong Kong	kh sql server	db	hk_rep_server

The three stages of subscription include creation, activation, and validation. Be sure to check the status of subscription following each stage:

1. The script *gensubdef.ksh* referred to above will automatically generate the subscription definitions for London database tables in Hong Kong. This script can be easily amended to create subscription definitions for any site.
2. Once the subscriptions have been defined, check their status by running *rs_helpsub* in the RSSD database for the rep server. This should show the status as defined.
3. Create and run a script called *activatesubdef.ksh* based upon *gensubdef.ksh*. Replace **DEFINE SUBSCRIPTION ${dbtable}_${DATABASE}_**[1] with **ACTIVATE SUBSCRIPTION ${dbtable}_${DATABASE}_**[1]
4. Run rs_helpsub again. This should show the status as active.
5. Create and run a script called *validatesubdef.ksh* based upon *gensubdef.ksh*. Replace **DEFINE SUBSCRIPTION ${dbtable}_${DATABASE}_**[1] with **VALIDATE SUBSCRIPTION ${dbtable}_${DATABASE}_**[1]

6. Run rs_helpsub again. This should show the status as valid.
7. Repeat the subscription definitions for other sites as well.
8. At the end of subscription definitions, you should have two subscription definitions for each replication definition. In other words, doing rs_helpsub for each table should give you 3x2 subscriptions = 6 lines. This should be shown in any RSSD database.

Use of Function Strings to Apply Local Timestamps

Replication Server converts functions to commands and submits them to destination data servers. For example, a new row inserted in the source table causes Replication Server to distribute an *rs_insert* function specific to that table to the subscriber databases. A possible solution for applying local timestamps at replicate database would be to modify *rs_insert* for a given source table to invoke an RPC at the destination database. The RPC in turn inserts local timestamp to the subscribed table. Note that for a peer-to-peer Replication System this process needs to be applied to all databases.

Handling Replication Maintenance

The rs_init facility automatically creates a maintenance login in the format of *database_name_maint*. This user is created as a public user in the replicate database. If you are using a warm standby set-up and your tables contain identity columns, you need to make user *database_name_maint* a "dbo" in the standby database, as this user needs to set the option *identity insert on* when replicating tables with identity columns.

Tuning Replication Servers for Better Performance

There are some configuration parameters that can be altered in order to get better performance from the Replication Servers:
init_sqm_write_delay: A stable queue manager waits for at least init_sqm_write_delay milliseconds for a block to fill before it writes the block to the correct queue on the stable device (default is 1000). Try decreasing this parameter.
init_sqm_max_write_delay: A flush to the queue is guaranteed to happen after waiting for init_sqm_max_write_delay, if the DSI or RSI thread reading the queue is unable to connect to the target or has been suspended (default 10000). Decrease this parameter if required. *sqt_max_cache_size:* You will need to increase this value if there are a lot of open transactions and or large transactions. Memory for sqt_max_cache_size is taken from the global memory pool (default is 131072 bytes).
batch_sz: The larger the batch_sz, the less often the truncation point is updated (default 1,000 commands).
This parameter is set in LTM configuration (cfg) file. Applicable to Replication Server up to 11.0.3.

Monitoring the Replication System

Replication Server provides a host of commands for checking replication status:

- Replication Server commands
- Replication Server Manager
- Sybase Central (Replication Server 11.5.1 and above)
- rs_subcmp program that allows comparison of tables in two replicate databases and has flags for reconciling them.
- Specifically written scripts

Beware of the use of *rs_subcmp* for reconciling large tables between remote databases. This may take a long time and will not always be practical. You may consider BCP'ing data instead. Also see the code posted with this article at www.isug.com for additional ideas.

Monitoring the Latency and Delivery of Data

Finally, it is a common practice for DBAs to set up a table in the replicated database where data is updated for the purpose of checking the latency and health of the replication system. In its simplest form, one can insert or update records in this table and see if the data is being replicated to the other sites. The time it takes for data to get to the remote site will give an indication of latency. However, it doesn't indicate whether the business transactions arrive in remote sites in a timely manner; nor does it allow for applications creating a large number of transactions where data delivery is impacted by concurrency, table size, or any locking mechanisms. It should also be noted that the maintenance user trying to deliver the replicated data may be blocked by local users. If the statistics on the user tables are not current, the replicated data may take a longer time to be delivered, resulting in remote users being blocked waiting for locks to be released.

Therefore, it is important to have a more realistic method for evaluating replication delivery. A possible solution is to look at the entries in an audit or transact history table in the replicate database and check the delivery timestamp. Any latency can be estimated by comparing the timestamps for records delivered and adjusting for server clock differences.

Syntax referred to in this article is available for download in the Members Only section of the ISUG website at www.isug.com. ■