Advanced Systems Lab

Milestone 1 Report

Karolos Antoniadis

1 Introduction

This report describes **mepas**, the system developed for "Advanced Systems Lab" project. Goal of this milestone was to create a message passing system platform supporting persistent queues and a simple message format. Furthermore to experimentally evaluate it and determine its performance characteristics. The desired message passing system consists of three tiers. The first one implements the persistent queues using a database, which form now on we will refer to as the "database tier" or "database". The second tier implements the messaging system and is responsible of all the logic related to system management, also it is the one tier that is using the database in order to implement its functionality. We will refer to this tier as the "middleware tier" or "middleware". Finally, the third tier that implements the clients that send and receive messages using the middleware, this tier is going to be referred to as "clients tier" or simply "clients". Figure [TODO] depicts the three tiers and they way they are connected to each other. As can been seen in the figure there is only one database while there can be more than one identical middlewares connected to the database. There can be many clients connection to different middlewares as well.

This report is written as follows: first we describe the general design of the system, including the database, the middleware and the clients. In next section we describe our experimental setup and how we conducted the experiments. At the nex

2 System Design

In this section we describe the design of the system. We start by describing the code structure of our system and the most important interfaces and aftewards we look more thoroughly at every tier and how its functionality was implemented.

Code Structure and Interface Overview

All the code for the client and the middleware was implement in subpackages of *ch.ethz.inf.asl*. Specifically the following spackages exist seen in the package structure of Figure 1.

While designing the system I came to the realization that the communication protocol, e.g. send message, receive message between the clients and the middleware could be the same between the middleware and the database. Because of this, the interface that can be seen in Figure 2 was created. This interface can be found in src/main/ ... and is implemented by both ClientMessagingProtocolImpl and MiddlewareMessagingProtocolImpl. The difference between the two implementations is that in the client implementation when for example sendMessage() is called, a message is sent from the client to the middleware that informs the middleware of the desire of the client to send

ch.ethz.inf.asl
- client :: contains classes related to client code
- common :: package containing common classes to be used by both the clients and the middleware
  - request :: contains all the possible request classes and the *Request* abstract class
  - response :: contains all the possible response classes and the general *Response* class
- console :: containts the management console code
- exceptions :: contains relevant exceptions used by the application
- logger :: contains the *Logger* class used for instrumenting the system
- main :: contains the Main class that is used to start the clients and the middleware
- middleware :: package containing classes related to the middleware
  - pool :: package containing pool implementations
    - connection :: contains the implementation of a connection pool
    - thread :: contains the implementation of a thread pool
  - utils :: contains general utility methods for the application

Fig. 1: Package Structure

a message. While on the other hand when the middleware calls sendMessage the middleware is calling a stored function from the database to actually "save" the message in the database.

```
int sayHello(String clientName);
void sayGoodbye();
int createQueue(String queueName);
void deleteQueue(int queueId);

void sendMessage(int queueId, String content);
void sendMessage(int receiverId, int queueId, String content);
Optional<Message> receiveMessage(int queueId, boolean retrieveByArrivalTime);
Optional<Message> receiveMessage(int senderId, int queueId, boolean retrieveByArrivalTime);
Optional<Message> readMessage(int queueId, boolean retrieveByArrivalTime);
int[] listQueues();
```

Fig. 2: Messaging Protocol Interface

MENTION that serializtion is used

Why are we having one Request/Response class per request/response?

It might seem weird having one request and one response class for every possible request and response. But this was done to make the code more extensible if needed to add new requests and for simplifying the work of the middleware. As can been seen the Request class contains the abstract execute method that receives as a parameter a MessagingProtocol implementation. Now when the middleware receives a request, after deserializing it it can just do 'request.execute(...)' and it knows that the correct execute method is going to be called. For example 'SendMessageRequest' implements execute as following. Therefore by taking advantage of polymorphism the Middleware

doesn't need a long and error-prone list of if-else like this 'if request is SendMessage do this ... else if ... '.

In order to create one more request for the system, the method has to be inserted in the MessagingProtocol and then implement it in ClientMessagingProtocolImpl and MiddlewareMessagingProtocolImpl and the correct REquest - Response classes to be implemented and that is it. There is no need to go around and introduce one more enum value or put one more 'else-if' at some part of the code.

Code conventions

Initially we were planning to run the system at Dryad cluster for testing purposes where we were said Java 7 would be installed. For this reason I re-implemented part of the Optiona class found in Java 8. Also almost everywhere the try-with-resources, a feature that appeared in Java 7 is being used so we can be assured that the close() is going to be called.

Database

Channging checkpoints cnfiguration file:

s; sync files=17, longest=0.035 s, average=0.006 s 2014-11-03 12:10:07 UTC LOG: checkpoint starting: xlog 2014-11-03 12:10:20 UTC LOG: checkpoint complete: wrote 1179 buffers (7.2%); 0 transaction log file(s) added, 0 removed, 3 recycled; write=13.352 s, sync=0.050 s, total=13.425 s; sync files=11, longest=0.012 s, average=0.004 s 2014-11-03 12:10:34 UTC LOG: checkpoints are occurring too frequently (27 seconds apart) 2014-11-03 12:10:34 UTC HINT:

—

Consider increasing the configuration parameter "checkpoint_segments".

——

checkpoint_segments = 1000 # in logfile segments, min 1, 16MB each checkpoint_timeout = 1h # range 30s-1h checkpoint_completion_target = 0.5 # checkpoint target duration, 0.0 - 1.0 checkpoint_warning = 30s # 0 disables

The PostgreSQL database management system was used, specifically PostgreSQL (release 9.3.5). It was need for the system to persistent store information so a database was used to store the needed information for the clients, the queues and the messages. For this reason three tables were created as can been seen in Figure 3 with their fields and their respective SQL types. As can been seen in this figure the fields *sender_id*, *receiver_id* and *queue_id* are all foreign keys of the *message* table. The first two are associated with the *id* of the *client* table, while *queue_id* is connected to the *id* of the *queue* table.

As can be seen all of the fields except the *receiver_id* of the *message* table cannot contain the *NULL* value. This was a deliberate choice since it is possible for a message to be sent with no particular receiver in mind and such a message could possibly be received by any other client (except the client that sent the message). In such a case, i.e. a message has no specific receiver, the *receiver_id* contains the *NULL* value.

The *message* table has also two check constraints associated with it. Those constraints are:

1. *CONSTRAINT check_length CHECK (LENGTH(message) <= 2000)*
2. *CONSTRAINT check_cannot_send_to_itself CHECK (sender_id != receiver_id)*

client
id *serial primary key*
name *varchar(20) NOT NULL*

queue
id *serial primary key*
name *varchar(20) NOT NULL*

message
id *serial primary key*
sender_id *integer REFERENCES client(**id**) NOT NULL*
receiver_id *integer REFERENCES client(**id**)*
queue_id *integer REFERENCES queue(**id**) NOT NULL*
arrival_time *timestamp NOT NULL*
message *text NOT NULL*

Fig. 3: Tables

The *check_length* constraint checks that a message cannot contain a message with too mich content, in this case one with more than 2000 characters. (TODO talk about text vs varchar) The second constraint was added because it was considered meaningless for a client to send a message to himself. It is also considered meaningless for a client to receive a message he sent (in case the *receiver_id* is *NULL*), this is also checked in the SQL functions and is expalined below (TODO).

In order to increase the performance of the database, indexes were used. PostgreSQL creates by default indexes on the primary keys[1]. The indexes that were introduced are the following:

1. *CREATE INDEX ON message (receiver_id, queue_id)*
2. *CREATE INDEX ON message (sender_id)*
3. *CREATE INDEX ON message (arrival_time)*

The first index was introduced to make faster the retrieval of message since most of them are based on a *receiver_id* and on a *queue_id*, *receiver_id* appears first on the index as its most commonly used[2] and in some cases its used alone, e.g. when listing the queues where a message for a user exists.. The second index was created to speed up receiving of messages from a specific sender. The third index was introduced since some of the receiving messages functions receive messages based on the arrrival time.

Code for the creation of the indexes and tables can be found in the `auxiliary_functions.sql` file in `src/main/resources`.

Stored Functions

Stored functions were created using the PL/pgSQL procedural language to reduce the network communication times between the middleware and the database. Also stored functions have the advantage that they are compiled already by the DBMS and their query plan has been generated

[1] "Adding a primary key will automatically create a unique btree index on the column or group of columns used in the primary key." (http://www.postgresql.org/docs/9.3/static/ddl-constraints.html)

[2] "...but the index is most efficient when there are constraints on the leading (leftmost) columns." (http://www.postgresql.org/docs/9.3/static/indexes-multicolumn.html)

and can be reused, therefore increasing performance. The code for the stored functions can be found in `read_committed_basic_functions.sql` file in `src/main/resources`.

Sta tests sinithos aplos calo ta statements etsi ... (den kalo ta callable statements) Stored procedures not used everywhere only where it made sense, e.g. management console.

Transactions and Isolation Levels

In this subsection we discuss isolation levels and why they are important for the correctness of our system. In order to do so let us see a simplified version of the internals of the *receive_message* stored function that takes as parameters the *p_requesting_user_id* and the *p_queue_id* and is looking for a message for the requesting user in the queue with the given id for receiving a message with no isolation levels in mind:

SELECT id INTO received_message_id FROM message WHERE queue_id = p_queue_id AND receiver_id = p_requesting_user_id LIMIT 1;

*RETURN QUERY SELECT * FROM message WHERE id = received_message_id;*

DELETE FROM message where id = received_message_id;

Functions in PostgreSQL are executed within transactions[3]. Transactions are known to be atomic, in the sense that they either happens completely or not at all. But still problems could arise as will be explained. The default isolation level in PostgreSQL is READ COMMITTED[4] which roughly states "...a SELECT query (without a FOR UPDATE/SHARE clause) sees only data committed before the query began; it never sees either uncommitted data or changes committed during query execution by concurrent transactions."[5]. So with such an isolation level it is possible for two concurrent transactions to read the same message, one of them to delete it and both of them will return the same message. This of course is not acceptable behaviour since we want a specific message ot be read by only one user. In order to solve this problem there are at least two approaches:

1. Use FOR UPDATE[6]and therefore preventing other transactions from selecting the same message.

2. Change isolation level to REPEATABLE READ which is stronger than READ COMMITTED and roughly states "This level is different from Read Committed in that a query in a repeatable read transaction sees a snapshot as of the start of the transaction, not as of the start of the current query within the transaction."[7]. In case another transaction deletes the message in the meantime the transactions is going to fail by giving back an error.

The "problem" with the second approach is that transaction could find concurrent update errors

[3] "Functions and trigger procedures are always executed within a transaction established by an outer query" (http://www.postgresql.org/docs/current/interactive/plpgsql-structure.html)

[4] "Read Committed is the default isolation level in PostgreSQL." (http://www.postgresql.org/docs/9.3/static/transaction-iso.html)

[5] http://www.postgresql.org/docs/9.3/static/transaction-iso.html

[6] "FOR UPDATE causes the rows retrieved by the SELECT statement to be locked as though for update. This prevents them from being modified or deleted by other transactions until the current transaction ends." and "hat is, other transactions that attempt UPDATE, DELETE, SELECT FOR UPDATE, SELECT FOR NO KEY UPDATE, SELECT FOR SHARE or SELECT FOR KEY SHARE of these rows will be blocked until the current transaction ends. " (http://www.postgresql.org/docs/9.3/static/sql-select.html#SQL-FOR-UPDATE-SHARE)

[7] http://www.postgresql.org/docs/9.3/static/transaction-iso.html

and will have to be re-executed[8]. For the above reasons I used the first approach since it made my application code easier, i.e. not having to repeat a transaction.

Talk about FOR UPDATE and ORDER BY — have a look

Connecting Java and PostgreSQL

For the connection between Java and the database the JDBC41 PostgreSQL driver[9] was used.

What is being logged?

Setting up the Database

This is coolness!!

Wha

Management Console

Middleware

The middleware implements the messaging system, it receives requests/messages from clients and has to use the database in order to persist those msssagess, as well as retrieve the messages from the database to return to the clients. Before explainging the general architecture of the middleware, the interface clients can use will be explained. This interface can be seen in Figure ?.

This interface satisfies the desired functionality of our system

The middleware follows a non-blocking approach using simple Java I/O. This seems hard to believe at first place but it is going to be explained later on. Before doing so, let us see some of the possible approaches that can be used to implement a middleware.

1. First approach would be to have some worker threads on the middleware which every one of them waits for a connection from the client. When a conneciton is established it waits for a arequest from the clients, when it receives the request it alls the middleware below it and returens the response to the client. Afterwards it closes the connection and waits for the next clent connection. This approach seems quite wasteful/slow since for every request-response interaction the client has to establish a connection with the middleware. Pseudocode of this solution can be seen:

2. Have a worker thread for every client. This solutions seems to have scalability issues since the number of clients the middleware could possibly handled is bounded by the number of conncurent ?threads the system can support.

3. Use Java New IO and use a selector thread ... blocking approach

4. My approach was to have a queue of sockets corresponding to connections for the clients and have some worker threads operating in a round-robin approach on those sockets and check if there is something to read from the underlying input stream of the socket. Benefits of this approach is that I can support more clients than the number of threads in my system, as

[8] "...it should abort the current transaction and retry the whole transaction from the beginning."(http://www.postgresql.org/docs/9.3/static/transaction-iso.html)

[9] http://jdbc.postgresql.org/download.html

well as avoiding to establish a connection for every request. + BufferedInputStream The non-blocking nature of this approach can be achieved by using InputStream's ? available method that can return the number of bytes that can be read whithout blocking. Available is also a non-blocking method. So a blocking read is called only when available showed that there are bytes available. Figure gives a graphical depiction of this approach. As can be seen.

Where do requests wait?

What is being logged?

Starting the Middleware

Many identical middlewares can be started
TAK ABOUT Implementing thread pool and connection pool bymyselfss

Stopping the Middleware

GRACEFUL termination

Clients

As was said in the previous section, clients use the 'MessagingProtocol' . Clients block when waiting for a response from teh middleware. How do they clients operate?
How do check the system is correct?
Talk about graceful termination

What is being logged?

Starting the Clients

3 Testing

How was the system tested? Mockito and testng. Mockito eg. to mock configuration files as well as connection ...
Testing Stored Procedures .. blah blah

End-to-End Tests

There are two end-to-end tests for our system. Both of them exist under the endtoend package. The first one exists in EndToEnd class while the other one in EndToEndWithMessages.

EndToEnd

This tests is as close as possible to how the system is going to be used. It creates two middlewares on the local machine and 4 clients also running locally, 2 clients connected to each middleware. The clients are beign executed for 20 seconds and they communicate with each other by sending and retrieving messages. At the end it is verified that number of requests sent by the clients were actually received by the middleware and that tha the number of reponses sent from the middlewares were actually received by the clients, those and only those. In order to check the requests and responses

that were being sent and received we had to inject some end-to-end testing code in the normal non-test code. This was done halfheartedly since it is a bad practice, like VisibileForTesting. But at the end it was worth it since after every change in the system by running this end-to-end test we could be assured that everything was still in place.

EndToEndWithMessage

f

Encountered Bugs

Verification errors while debugging .. forgotten notNull Found the InstantiationException in the newInstance() thing. Because I had requests with not a nullable Constructor. (This was found while mocking to get the messages with failed response).

While writing the endtoend test I realized I was immediately closing the connection to the clent from the middlware when the client was saying goodbye so the user was waiting forever for a response from the middleware. I though it was the middleware that wasn't finishing in the test so I Started making all the thread daemon threads to see what will happen. Found bugs in equals methods ...

General Encountered Problems

Problems that werent bugs

4 Experimental Setup

about properties files (maybe) :

Sunday(19/10/2014) ———————— By changing the executable to read configuration files instead of the command line arguments I solved the '$' problem in the password and also I can chaange the configuration files without having to really change the deployment scripts!!

Talk about pexpect and how awesome it is!

EC2 Instances

Do the following http://superuser.com/questions/331167/why-cant-i-ssh-copy-id-to-an-ec2-instance ssh-add privatekye file to login to the ec2 instances without having to do 'ssh -i ~/... ' every time!! AWESOME!!

Deploying the System for Experiments

Explaing about Python boto .. gnuplot and all the other packages ... most of the time used UNIX native commands to have much faster times of whaterver ... for example for reading CSV files, cutting parts of files and so on

5 Experiments

All successfull receiveals bla blah

——— In the report mention that in the throughput all the requests were successful, I had no failed responses.

Stability

Clients and MW was t2.small and db t2.medium.

As can be seen in Figure ... Used getTrace method from ResultsReader.py to extract the data. Response time was averaged over the interval of one minute. While throughput was calculated per second and averaged over the interval of one minute. I.e. In minute i corresponds to the averaged time from (i - 1, i]. The data were generated using the getTrace from ResultReader

Let's see where time was spent

adfs	dsafasdf
CONNECTION: 0.000689706, 0.216034	
REQUEST: 3.23399, 2.38918	
IN WORKER THREAD QUEUE: 13.3207, 4.17608	
TIMES A SOCKET IS WORKED for a REQUEST TO BE READ: 1, 0	

TIMES TO ENTER: 1.00164, 0.163046 TIMES (NOTHING) INSIDE: 0.0201363, 0.570286 TIMES (DOING) INSIDE: 3.33539, 2.51786

Averages and SD

25 17.10725 5.51819 RECEIVE MESSAGE

25 17.1355 5.543025 SEND MESSAGE

25 15.7621 4.689385 LIST QUEUES

16.668 average response time in total Gia ola ta requests

Talk about list queues(Since this was the trace and we just wanted to verify that our system is stable when it is being executed for a fair amount of time we did not really do any assumptions about the results. There are some things that actually can be asily explained, the time to receive a connection is technically 0, this is because we have the same amount of worker threads to connections. The time waiting for a worker thread is quite high and was to be expected since we have 100 clients and only 20 workwer threads. So at any point in time 80 clients connections could possilby be waiting. Network time is also really low and this makes sense since the throughput between two instances is (iperf). List queues check db time and why they are faster, not doing so much with the database ... verify this by checking request time for all types of requests.

DB REQUEST per type of request

grep "DB REQUEST\tLIST_QUEUES" middlewareInstance1/m*.csv | awk -F'\t' '{ sum += $2; n++; } END { if (n > 0) printf sum /n }' 2.38218

grep "DB REQUEST\tSEND_MESSAGE" middlewareInstance1/m*.csv | awk -F'\t' '{ sum += $2; n++; } END { if (n > 0) printf sum /n }' 3.71801

grep "DB REQUEST\tRECEIVE_MESSAGE" middlewareInstance1/m*.csv | awk -F'\t' '{ sum += $2; n++; } END { if (n > 0) printf sum /n }' 3.60231

TIME WISE (for all request ... < 50ms are 99.7% of the requests

< 50 0.997697

< 25 0.97127

< 20 0.869102

< 23 0.952159
< 22 0.9357

$2^{k(=?TODO)}$ Experiment

Before starting I would like to talk about love the one and only one.
Factors
number of middleware threads
number of connections
instance type of middleware
instance type of database
dfs
16 experiments in total

Increasing the Message Size

Increasing the Number of Clients

6 Conclusion

This is a lovely conclusion for a lovely world that used to exist but is no more.

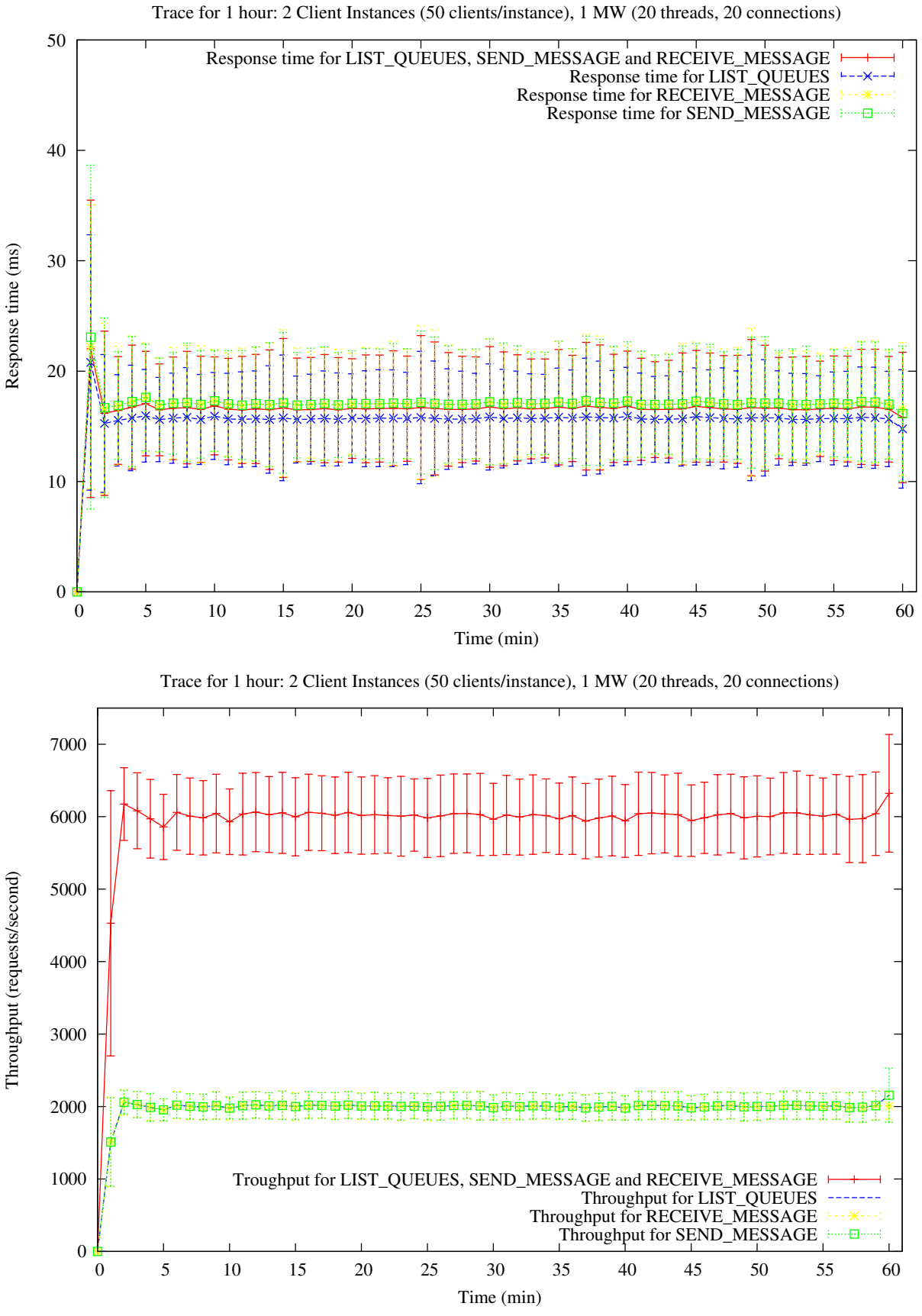

Fig. 4: Response time and throughput of an one hour trace with 2 client instances (50 clients/instance) and 1 middleware instance (20 threads, 20 connections)