# Morgan Stanley’s Global Trading Placement Engine Challenge

Team Size 2-4

Due May 5, 2013

Introduction

In today’s increasingly global and fast-moving trading world, banks need to react ever-faster to changing business opportunities, and deploy technology to address client demand whenever and wherever it occurs. This **Global Trading Placement Engine** challenge gives you a practical introduction to some of the challenges in this trading world, and presents you an opportunity to showcase your critical thinking, team-working and programming skills.

In previous application deployment models in our Bank, applications have been deployed on a fixed number of physical servers, in specific locations (e.g. New York City). Whilst this works well for local clients of the application, those in different continents (e.g. Europe, Asia-Pacific) receive a poorer service experience. Worse still, if the demand increases to saturate the fixed number of application servers, then everyone gets a poor experience.

In order to address these problems, newer applications are adopting component-based ‘Service Oriented Architecture’ (SOA) design patterns, where the application is decomposed into multiple different tiers, each running on physical or virtual servers. A typical deployment pattern would be a client desktop using a modern browser, talking to a web-server tier (e.g. running Apache), which talks to a business logic tier (e.g. running Java), which talks to a database tier (e.g. running IBM DB2). Each of these tiers can then be configured to run as a resource pool, by leveraging load balancers, J2EE servers, or parallel databases and database replication. Added benefits come from deploying these tiers using Virtual Machines instead of on the physical servers, as this provides greater flexibility and agility.

## The Challenge

For this challenge, we want to introduce you to a fictitious trading application called Hermes (named after the Greek god of boundaries, travel, communication and trade!!). This fictitious application runs in North America (NA), Europe (EU), and Asia-Pacific (AP), and is built on a three-tier SOA architecture. Hermes handles transactions from clients on a 24x7 non-stop basis. The application has a Web layer, a Java layer and a Database layer, comprised of virtual machines running in the three trading regions.

Your challenge is to write a control algorithm to control how many virtual machines should be running Hermes at any moment of the day, and in which locations. To showcase your algorithm, you will compete in a trading simulation, collecting the trading transaction volumes and the infrastructure configuration across a network sockets interface, and displaying the results on a GUI or Web site of your choosing.

Don’t worry! We will explain everything you need to know to be successful with this challenge.

## Deliverables

1. A 5 minute presentation (in any format you prefer), describing how you approached the challenge, and any key features of the application or the algorithm that you particularly want to highlight.
2. Your application comprising three main pieces of code:
   1. A GUI to show what is happening with your algorithm, and the status of the simulation.
   2. Network sockets code to communicate with the judge’s master server which feeds you the simulation details.
   3. A control algorithm to run the logic of your controller.
3. Source code to your application, showcasing your programming and documentation style.

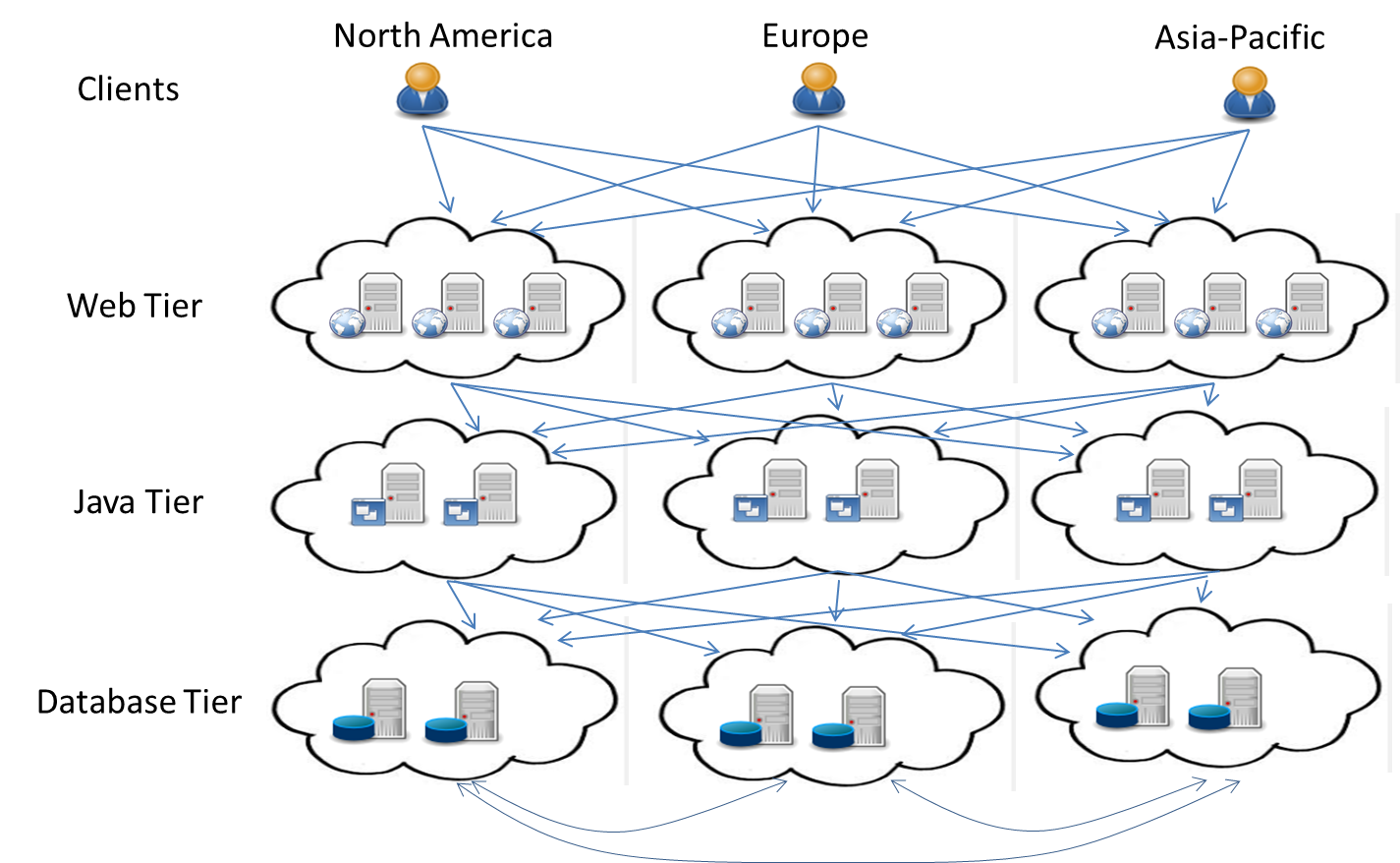
## Judging criteria

1. Successful run of your application against the judges’ master server, completing the network conversation as documented below, and updating your GUI accurately.   
   **20% weight**
2. The visual appearance, clarity, accuracy of your GUI.  
   **20% weight**
3. The trading profit from a completed run of the game.  
   **20% weight**
4. The style and clarity of your source code.   
   **20% weight**
5. Your presentation.  
   **10% weight**
6. The X Factor – anything unique about your solution that the judges particularly liked.   
   **10% weight**

## Conditions

* Your application can be written in a language of your choice – e.g. C++, Java, Python – and run on an OS and machine of your choice (e.g. PC, VM, Amazon EC2). The only requirement is that it will need network connectivity to the judges’ PC or server.
* You are free to include any GPL or LGPL open-source code into your application as you wish, so long as you fully credit the work to the owner, and you make clear what is yours and what is copied.
* Your code has to be labeled as GPL or LGPL open-source.

## The Hermes Application Simulation

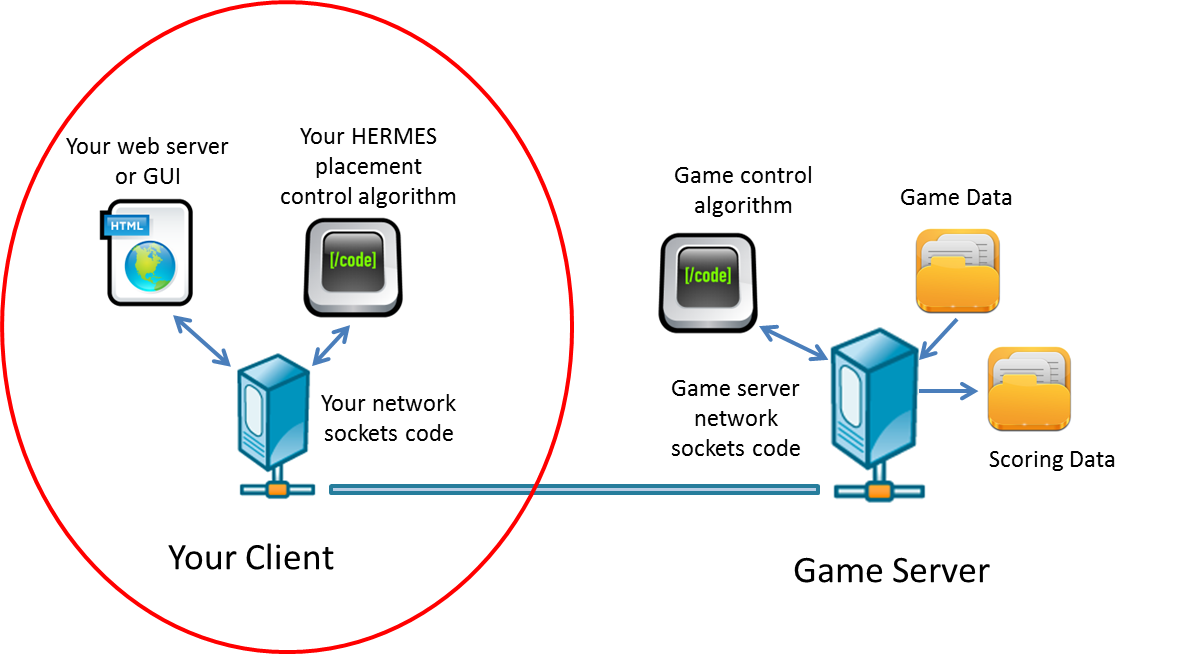


This is a schematic diagram of the Hermes application. Clients access Hermes from North America (NA), Europe (EU), and Asia-Pacific (AP) at any time of the day or night. They access a web front-end hosted on one of several web servers in the Web Tier. As the application is quite interactive, the latency between the client and the Web Server is important. Wherever there is spare capacity, the web load balancers will distribute the client load locally (i.e. AP clients to AP web servers, and same for EU and NA). If the local web servers approach capacity, the load balancers will automatically redirect the traffic over onto an out-of-region web server (if there is spare remote capacity). As this happens, the application will lose some transactions (due to clients not accepting the slow application and either not trading or going to a competitor). If the client traffic cannot be redirected, then it will be dumped onto the local web server. When this happens, the local web servers will become extremely slow, and many transactions will be lost. This is a bad thing! As the web servers run independently, you should think about running web servers in each location able to handle most of the traffic locally.

For the transactions that are successfully handled on the Web Tier, they will get distributed to the Java Tier for processing. The locality of the Java server to the web server is somewhat less important than the client to the web, but it’s still important to align the demand if possible. Like the web servers, the Java servers also run independently, so you can distribute these to the three locations to match the local demand.

In the course of processing the transaction, the Java server will talk to the Database Tier to access trading data (e.g. prices, stock symbols, etc), and to place the trade. The database tier is less sensitive to location than the Java servers. Within a single site, you can run multiple instances of the database server as needed to handle the load. It’s a parallel database, so it scales approximately linearly with load. The database needs to synchronize its state across all instances, so whilst this is treated as a negligible performance penalty when the instances are in a single site, you will see a performance penalty running the database stretched across two sites, and a much larger penalty if you try to stretch it across all three sites. You should think carefully about where you want the database to run (given the current load conditions), and when moving it, start instances on the new site and then shut down on the old site, so as to minimize this cross-site database performance penalty.

## The Global Trading Placement Engine environment



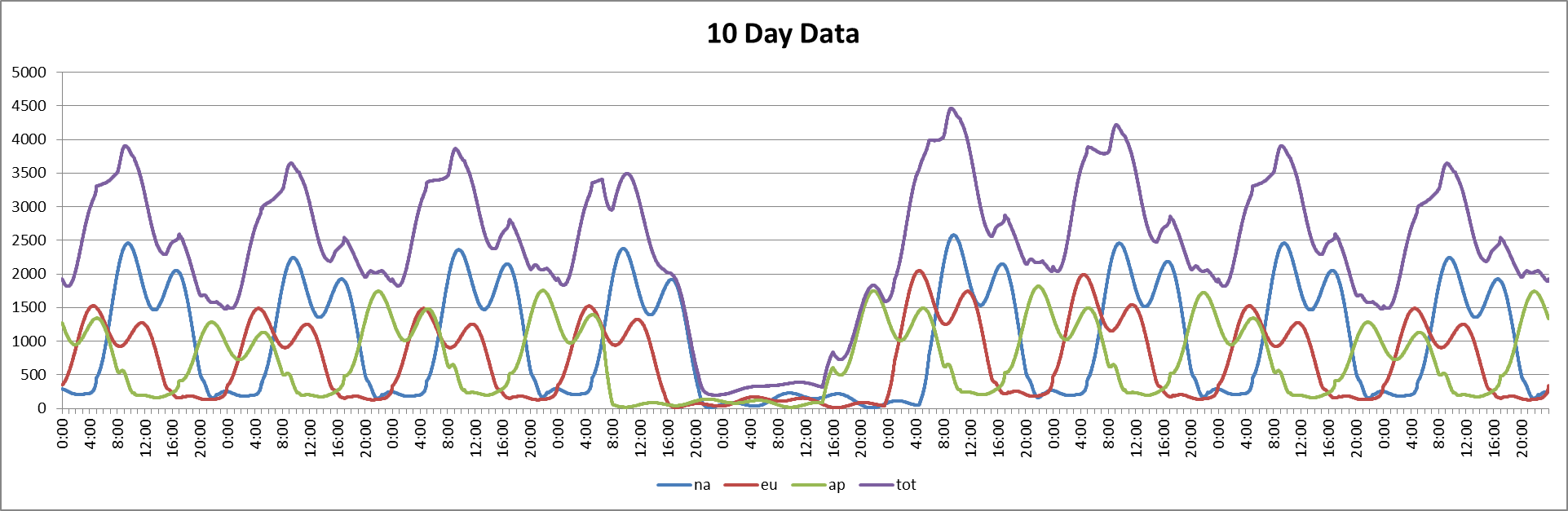
This is a schematic diagram of the Global Trading Placement Engine environment that you will be coding. You will be writing an application comprising three major components:

1. A network sockets component to communicate with the Game Server.
2. A GUI or Web Server to display the game progress.
3. Your HERMES placement control algorithm, where you will decide how many web servers, Java servers and database servers you wish to run in each site at any point in the simulation.

The Game Server will run on a central PC, laptop or server. It communicates to you through the network sockets interface. It reads game data from a control file, executes the game control algorithm, and puts the resulting scores into a file for later judging.

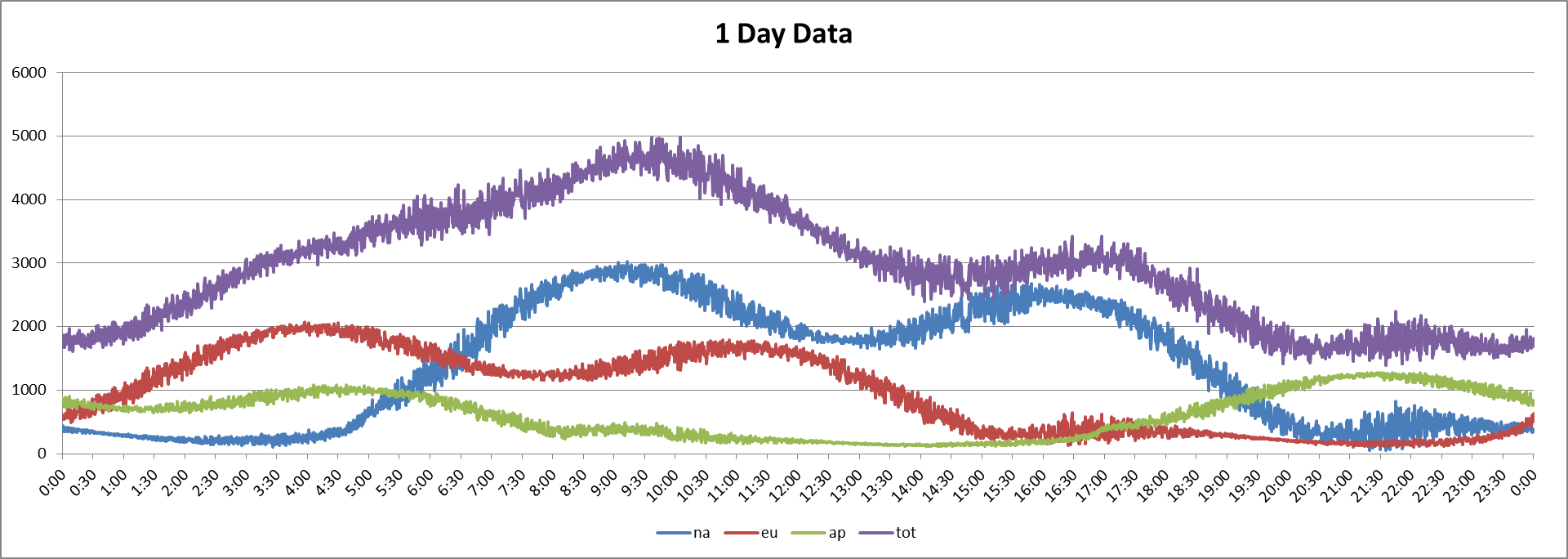
The game is a turn-based simulation. Each turn represents 30 seconds of trading activity. On each turn, you will receive the current configuration (i.e. how many web, java and database servers are running in each site), the client transaction demand for the previous period (split into three demand numbers, one per site), the distribution data (showing how the demand was distributed to each tier and location), and the profit or loss for that period. Your job is then to send back your desired control inputs, stopping or starting a number of web servers in a location of your choice. The game simulates between 6 hours and 2 weeks of trading activity, always in 30 second intervals.

Here is a sample of the underlying trading pattern:



You will notice that each day has a morning peak at around 9am, and an afternoon peak at around 4pm. The EU peaks occur 5 hours earlier than the NA peaks (due to time zone), and are slightly smaller than the NA peaks. The AP peaks happen 7 hours before the EU peaks, and are usually slightly lower than the EU peaks. You can also see what happens at the weekend, with AP dropping close to zero first, then EU, then NA. Note that for any specific day, the volumes can be higher or lower depending on market conditions.

You may assume this underlying pattern for your control algorithm (with daily differences in peak sizes per region). However, at a more granular level, there are peaks and troughs all the time, as shown here:



## Game Parameters

This section describes the game parameters, the load distribution rules, and the network sockets protocol you need to code to communicate with the game server.

Note that all game parameters are integers in this simulation.

REVENUE Cents of revenue per transaction.

W\_COST Dollars of cost per web server per 30 second ‘game turn’

J\_COST Dollars of cost per Java server per 30 second ‘game turn’

D\_COST Dollars of cost per database server per 30 second ‘game turn’

CONFIG (9 ints) This is the current configuration of Hermes. The numbers are:

W.na (i.e. Count of Web servers in North America),

W.eu (i.e. Count of Web servers in Europe),

W.ap (i.e. Count of Web servers in Asia-Pacific),

J.na (i.e. Count of Java servers in North America),

J.eu (i.e. Count of Java servers in Europe),

J.ap (i.e. Count of Java servers in Asia-Pacific),

D.na (i.e. Count of Database servers in North America),

D.eu (i.e. Count of Database servers in Europe),

D.ap (i.e. Count of Database servers in Asia-Pacific)

E.g. 0 2 0 3 1 0 0 0 2 would represent 2 web servers in EU, 3 Java in

NA, 1 Java in EU and 2 database servers in AP.

DEMAND (Day HH MM SS then 3 ints)

e.g. MON 14 30 30 203 599 1277

This is the actual transaction demand on Hermes in the previous

30 seconds. The day and time are in Eastern Time. The numbers are T.na, T.eu, T.ap. The example represents 203 transactions in NA, 599 in EU and 1277 in AP.

DIST (9 ints) This represents the distribution of the last transaction demand.

The format is an integer number of successful transactions on

each tier and site. This information is provided for you to see how

game server is distributing the load on a transaction level.

Distribution rules are explained in the next two sections.

PROFIT (4 ints) Profit is returned in a set of 4 integers, representing the profit in the

last period, the percentage of profit to max revenue possible in the

last period, the total cumulative profit to date, and the percentage

of profit to the max revenue possible. Note that you can never achieve 100% profit, as you will always need to pay for some infrastructure to execute the transactions.

CONTROL (9 ints)

E.g. 0 0 1 -3 0 0 0 1 1

This is the control input you generate to send to the game server.

The format represents the desired changes to W.na, W.eu, W.ap, J,na, J,eu, J,ap, D.na,  
D.eu, D.ap. The example represents a command to add a web server to AP, remove 3 Java servers in NA, and add a database server to each of EU and AP.

## Understanding the distribution and processing limits

You need to adjust the number of Web, Java and Database servers running to accommodate the changing transaction workloads in each location. This sections discusses the limits on each component, and how you ca optimize the solution.

Each web server has the following processing limits each 30 seconds:

|  |  |  |
| --- | --- | --- |
| Web Server Performance Table | | |
| Transactions | Successful | Comments: |
| 0-180 | 100% | Nominal web performance. All successful. |
| 181-200 | 90% | Web server “at capacity”: 10% of transactions being lost. |
| 201-250 | 20% | Web server overloaded: 80% loss. |
| 251+ | 0% | Web server hung: 100% loss. |

You should try hard to not exceed 200 transactions per Web server per game turn, as you can see that performance really dives at this point. If you exceed 250 transactions per Web server, then you will lose all the workload from that server, as the performance is considered so poor that the transactions will all be canceled or rejected.

Each Java server has the following processing limits each 30 seconds:

|  |  |  |
| --- | --- | --- |
| Java Server Performance Table | | |
| Transactions | Successful | Comments: |
| 0-400 | 100% | Nominal Java performance. All successful. |
| 401-500 | 90% | Java server “at capacity”: 10% of transactions being lost. |
| 501-600 | 20% | Java server overloaded: 80% loss. |
| 601+ | 0% | Java server hung: 100% loss. |

As you can see, the Java servers perform similarly to the Web servers, but with higher transaction thresholds. As with the Web server, try to keep the load below 500 per instance, as performance steeply degrades at this point.

Each database server has the following processing limits each 30 seconds:

|  |  |  |
| --- | --- | --- |
| Database Server Performance Table | | |
| Transactions | Successful | Comments: |
| 0-1000 | 100% | Nominal database performance. All successful. |
| 1001-1200 | 90% | Database server “at capacity”: 10% of transactions being lost. |
| 1201-1400 | 20% | Database server overloaded: 80% loss. |
| 1401+ | 0% | Database server hung: 100% loss. |

The databases also look similar to Web and Java, but with the highest thresholds.

The next table explains what happens as you distribute client transactions on the web tier:

|  |  |  |
| --- | --- | --- |
| Client Distribution Performance Table | | |
| Client-to-Web | Successful | Comments: |
| Same Region | 100% | Nominal web responsiveness. (App feels like client expects.) |
| EU to/from NA | 70% | 20% loss for Trans-Atlantic web serving. (App feels slow.) |
| AP to/from NA | 50% | 50% loss for Trans-Pacific web serving. (App feels really slow.) |
| AP to/from EU | 50% | 50% loss for Trans-Asia web serving. (App feels really slow.) |

As you can see, there is a penalty for satisfying the transactions on an out-of-region Web server. The application will feel slow, and some of the transactions will be lost as the clients go to competitor’s solutions instead of ours. You will need to balance the cost of running infrastructure everywhere versus the lost opportunities from running from an out-of-region location.

The out of region losses for the web – to – Java connections are as follows:

|  |  |  |
| --- | --- | --- |
| Web Distribution Performance Table | | |
| Web-to-Java | Successful | Comments: |
| Same Region | 100% | Nominal Java responsiveness. |
| EU to/from NA | 90% | 10% loss. |
| AP to/from NA | 80% | 20% loss. |
| AP to/from EU | 80% | 20% loss. |

In contrast to the Client – to – Web servers, where the responsiveness is more sensitive to the client, the Web – to – Java is slightly less sensitive to location. As you can see, you lose 10% of transactions going Trans-Atlantic, and 20% going Trans-Pacific. You need to think about whether or not to run a more centralized Java plant, or whether to distribute it.

The out of region losses for the Java – to – database connections are as follows:

|  |  |  |
| --- | --- | --- |
| Java Distribution Performance Table | | |
| Java-to-DB | Successful | Comments: |
| Same Region | 100% | Nominal Database responsiveness. |
| EU to/from NA | 90% | 10% loss. |
| AP to/from NA | 90% | 10% loss. |
| AP to/from EU | 90% | 10% loss. |

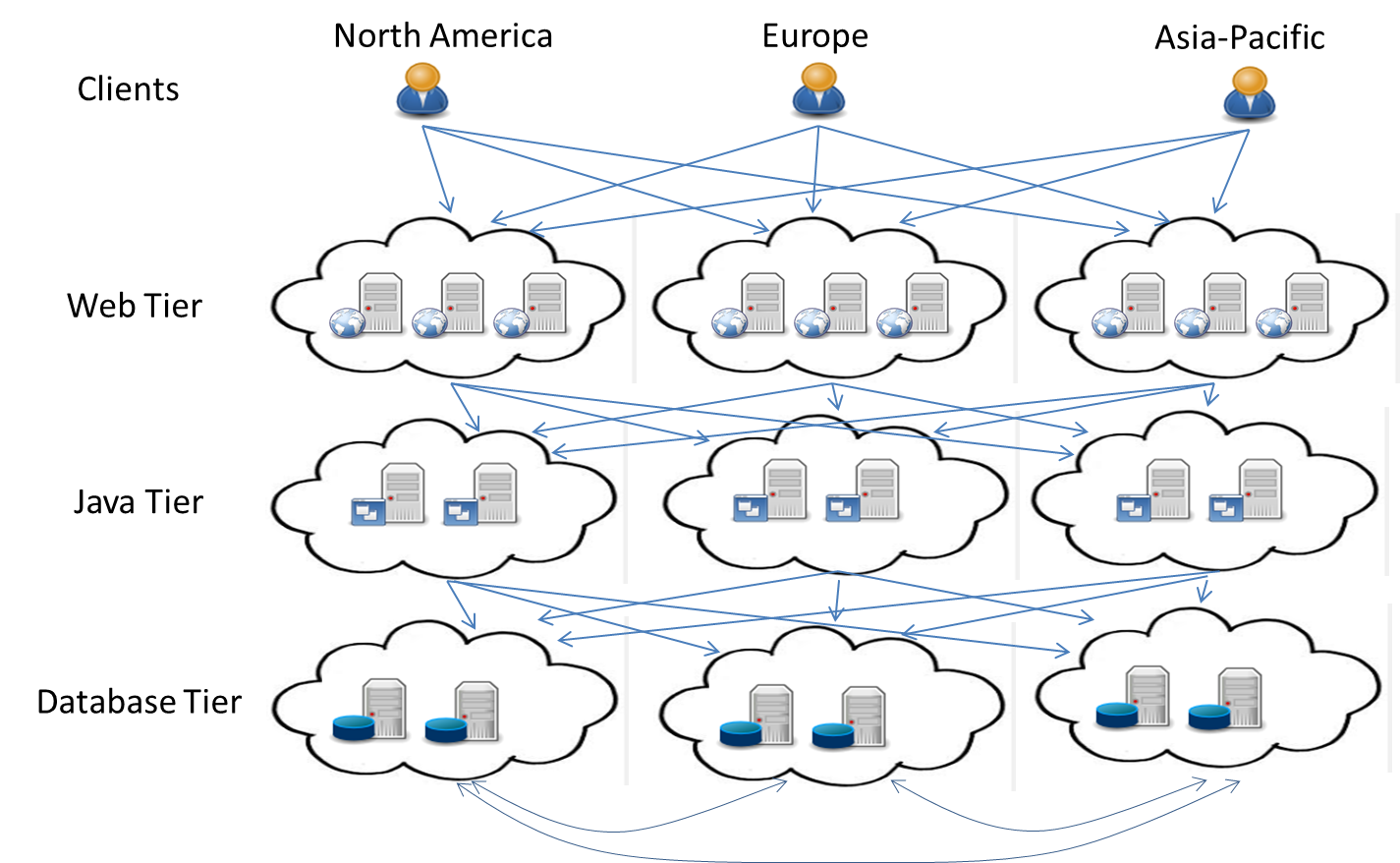
The Java – to – Database distribution is relatively benign. You only lose 10% for running out of region, Trans-Atlantic or Trans-Pacific. This indicates a stronger preference to run one region for the database at any one time, instead of multiple regions.

|  |  |  |
| --- | --- | --- |
| Database Split-Site Performance Table | | |
| DB loss | Successful | Comments: |
| Same Region | 100% | No losses for running inside one region. |
| EU to/from NA | 80% | 20% loss due to DB synching trans-Atlantic. |
| AP to/from NA | 70% | 30% loss due to DB synching trans-Pacific. |
| AP to/from EU | 70% | 30% loss due to DB synching trans-Pacific. |
| Triple-site | 39% | = 80% \* 70% \* 70% success rate. Worst overhead possible! |

This table emphasizes the point about running in a single region. As soon as you run the database across two regions, the synchronization and transaction-commit overhead starts to hurt performance. In the worst case, where the database is concurrently active in three sites, you will suffer a compounding effect of trying to three-phase commit the transactions globally. Don’t do this if you can at all help it.

Summary of Loads and Success Rates

Here’s a graphical summary of all those transaction loads and distribution success rates:



0-1000 = 100%, 1001-1200 = 90%, 1201-1400 = 20%, 1401+ = 0%

0-180 = 100%, 181-200 = 90%, 201-250 = 20%, 250+ = 0%

0-400 = 100%, 401-500 = 90%, 501-600 = 20%, 601+ = 0%

70%

70%

80%

90%

90%

90%

90%

100%

100%

100%

80%

80%

80%

90%

100%

100%

100%

50%

50%

50%

100%

70%

100%

100%

## Understanding the distribution, processing, and profit logic

In the game server, the following steps are applied to distribute and process the transactions, and to generate the game profit. (Note – this is all calculated by the game server, so this section is just for interest, as you do not need to code it yourself.)

1. The Client transactions are distributed as follows, until all transactions are distributed:
   1. Local transactions go to local web servers, up to the “at capacity” level.
   2. Any remaining goes to the nearer of the other two sites (subject to the cross-site distribution losses as described above), up to that site’s “at capacity” level.
   3. Any remaining goes to the further of the other two sites (with distribution loss per the Client Distribution Performance Table above), up to that site’s “at capacity” level.
   4. All remaining is dumped on the local web tier (even if this overloads it or if the local web servers are not running).
2. The Web tier processes the transactions at each site as follows:
   1. If no servers, then the loss is 100%.
   2. Divide the load by the number of web servers.
   3. Apply the appropriate processing loss from the Web Server Processing Table.
3. The remaining web transactions then are distributed to the Java tier, applying the logic in step 1 and the Web Distribution Performance Table.
4. The processing loss on the Java tier then is calculated per site, applying the logic in step 2 and the Java Server Performance Table.
5. The remaining Java transactions then are distributed to the Java tier, applying the logic in step 1 and the Java Distribution Performance Table.
6. The processing loss on the database tier then is calculated per site, applying the logic in step 2 and the Database Server Performance Table.
7. If the database is running on more than one site, then apply the processing loss according Database Split-Site Performance Table.
8. The profit is the remaining transactions times the profit per transaction, less the cost of each web server, each Java server and each database server. To maximize the profit, you need to process the maximum number of transactions from the minimum amount of hardware.

## The GUI or Web interface

## You need to provide a visual indication of the progress of your application as it runs the game. You have very wide discretion in how to present this information. At the simplest, you could simply stream information into a scrolling window. At the most complex, you may wish to have a fully functioning web interface, with movable web-parts, real-time graphing, and your game control and algorithm tuning parameters all on the screen for interaction.

There are some key items, though, that we would like you to be able to display clearly. These are:

1. The current demand per region.
2. The current configuration counts of the various web, java and database components.
3. The current and running total profit, expressed in dollars, and in % of total revenue potential.

Present these in numerical format somewhere on your GUI.

Optional, but useful other elements would be:

1. A graph of the demand versus the supply for each tier or each location, or for individual tier and location components.
2. Some indication of the current versus desired state for your algorithm, driving your control inputs.
3. Some way to stop/start/slow down /speed up/ reset your game from the GUI.
4. In the case of any error you gracefully catch (e.g. a sockets error), present the error description on the screen, and give the user the option to reset and restart the game.

## The Network Protocol and Conversation Map

You need to communicate with the Game Server over TCP/IP using network sockets programming. If you have not done this before, then these two URL’s may be of use for you:

For Linux: <http://codebase.eu/tutorial/linux-socket-programming-c/>

For Windows: <http://www.madwizard.org/programming/tutorials/netcpp/3>

The basic steps that you client needs to do are as follows:

1. (Windows only) Initialize the WinSock environment with WSAStartup();
2. Create a socket with socket();
3. Connect to the Game Server, with connect();
4. Send data to the Game Server with send(); calls as needed.
5. Receive data from the Game Server with recv(); calls as needed.
6. Shutdown your connection with shutdown();
7. Close the socket with close() or closesocket();
8. (Windows only) Shutdown the WinSock environment with WSACleanup();

Your actual commands and syntax of course vary depending on language and operating system, but this should get you going.

You are required to follow this conversation map for the actual game protocol.

You open the connection

You: INIT {teamname}

Game: ACCEPT

You: RECD

Game: COSTS {profit} {W-cost} {J-cost} {D-Cost}

You: START

Game loop:

Either you get …

Game: CONFIG {W-NA} {W-EU} {W-AP} {J-NA} {J-EU} {J-AP} {D-NA} {D-EU} {D-AP}

Or you get …

Game: END

If end … you jump out of the loop and finish with STOP

You: RECD

Game: DEMAND {DDD} {HH} {MM} {SS} {NA-U} {EU-U} {AP-U}

You: RECD

Game: DIST {W-NA} {W-EU} {W-AP} {J-NA} {J-EU} {J-AP} {D-NA} {D-EU} {D-AP}

You: RECD

Game: PROFIT {profit} {%of max} {cumulative\_profit} {% of max cumulative}

You: CONTROL {W-NA} {W-EU} {W-AP} {J-NA} {J-EU} {J-AP} {D-NA} {D-EU} {D-AP}

Go back to game loop…

You: STOP

You shut the connection.

The details of each of these Game Parameters was defined previously in this document.

Here’s a sample command line trace between a testing game client and the real game server, with added commentary for your review:

C:\> Hermes\_test\_control\_client localhost 22222 Bravo *Command Parameters: host, port, team name*

Initializing sockets interface… initialized. *Initialize the sockets interface (i.e. WinSock on Windows)*

Initializing connection…

Looking up hostname localhost… found. *DNS name lookup … to find 127.0.0.1*

Creating socket… created. *Calling socket()*

Attempting to connect to 127.0.0.1:22222… connected. *Calling connect()*

… initialized.

Starting conversation.

>> INIT Bravo *No quotes needed. (The >> and << are also not sent or received)*

<< ACCEPT

>> RECD

<< COSTS 9 3 6 15 *This says $0.09 revenue per transaction, $3, $6, $15 for each*

>> START *web, java, database instance respectively.*

<< CONFIG 2 2 2 1 1 1 0 1 0 *First config (could have been an END). 6 web servers (2 in NA, 2*

>> RECD *EU, 2, AP), 3 Java, 1 DB in Europe.*

<< DEMAND MON 00 00 00 176 248 621 *First demand … it’s Monday at midnight and most demand is in AP*

>> RECD

<< DIST 178 400 400 178 360 360 0 844 0 *Distribution data. You can see the AP web traffic is spilling over to*

>> RECD *saturate the EU web instances and a tiny bit to NA.*

<< PROFIT 24 25 24 25 *We made $24 profit = 25%. That’s good!*

>> CONTROL -1 0 2 0 0 1 0 0 1 *Test algo kicks in. Rebalance Web, Java and DB. Note – don’t shut*

*down the EU database until the AP one is online, or we will lose*

*all transactions!*

<< CONFIG 1 2 2 1 1 1 0 1 0 *Note timing on stops and starts. Web stops immediately, but needs*

*two turns to start. Longer stop/starts for Java and Database.*

>> RECD

<< DEMAND MON 00 00 30 176 245 595 *Monday 00:00:30 now.*

>> RECD

<< DIST 176 381 400 176 342 360 0 824 0

>> RECD

<< PROFIT 3 3 27 14 *Temporarily less profit as the database is in transition (losing transactions*

*as we are trying to do trans-pacific synchronization!)*

>> CONTROL 0 0 0 0 0 0 0 0 0

<< CONFIG 1 2 2 1 1 1 0 1 0

>> RECD

<< DEMAND MON 00 01 00 164 249 593 *Monday 00:01:00 now.*

>> RECD

<< DIST 164 384 400 164 345 360 0 816 0

>> RECD

<< PROFIT 3 3 30 10

>> CONTROL 0 0 0 0 0 0 0 0 0 *Note – you don’t need to actively control every turn … send some zeros!*

<< CONFIG 1 2 4 1 1 1 0 1 0

>> RECD

<< DEMAND MON 00 01 30 165 235 617 *Monday 00:01:30 now.*

>> RECD

<< DIST 165 235 617 165 340 500 0 893 0

>> RECD

<< PROFIT 2 2 32 8

>> CONTROL 0 0 0 0 0 0 0 0 0

<< CONFIG 1 2 4 1 1 1 0 1 0

>> RECD

<< DEMAND MON 00 02 00 171 242 616 *Monday 00:02:00 now.*

>> RECD

<< DIST 171 242 616 171 346 500 0 904 0

>> RECD

<< PROFIT 2 2 34 7

>> CONTROL 0 0 0 0 0 0 0 0 0

<< CONFIG 1 2 4 1 1 2 0 1 0

>> RECD

<< DEMAND MON 00 02 30 176 232 594 *Monday 00:02:30 now.*

>> RECD

<< DIST 176 232 594 176 232 594 0 924 0

>> RECD

<< PROFIT -2 -2 32 5

>> CONTROL 0 0 0 0 0 0 0 0 0

<< END *And END !! For this sample, we only ran for 2:30 mins. Normal runs are*

*between 1 and 10 days, so expect thousands of opportunities*

*to send in control commands.*

>> STOP

Conversation completed successfully. *Shutdown()*

Closing connection and cleaning up socket interface… *Close() or CloseSocket(), and we are done! You may wish to store the data*

Successful completion. *locally so you can replay the game on your web/GUI interface, or zoom on*

*specific periods. You may also need it so you can tune your algorithms.*

## Discussion on the Hermes Placement Control Algorithm

This section will give you some ideas and things to think about with your control algorithm. By now, you will have realized that you are trying to find the optimal number of web, Java, and database servers to handle the changing demand, and you have to also think about where to run these components.

You first need to know the following start up and shut down times for the various components:

|  |  |  |  |
| --- | --- | --- | --- |
| Tier | Start Time | Stop Time | Comments |
| Web | 60 secs | 0 secs | Two game turns to start, immediate stop. |
| Java | 120 secs | 30 secs | Slower start than web, one turn to stop. |
| Database | 240 secs | 60 secs | Slower start and stop than Java tier. |

This is important if you want to take action on every turn, as you need to remember that some components will not start for up to 8 game turns. You also need to be careful to confirm that your new component is started before you switch off your old component, or you can have game turns with nothing active. For example, if you choose to move your database from AP to EU and you do the start and stop on the same turn, then you will have 6 game turns of no database server running, because the old database servers will stop in 60 secs and the new ones take 240 secs to start!

So what’s a simple starting point? How about looking at your configuration once every 10 minutes, and average the demand per location for those 10 minutes, and decide what to adjust. This will not be the most optimal of algorithms, but it avoids having to make a note of the pending startup and shutdown actions.

You can process the DIST game parameter if you wish, in order to gain insight into the distribution of the load through the various tiers. Or, you can control the game simply from the DEMAND game parameters.

You should remember that the game has a distinct underlying daily and weekly pattern. Your demand data is giving you the day and time, so you could code an algorithm to lock onto the underlying signal if you wanted to get more sophisticated.

You may wish to search for “PID” control algorithms, which form the basis for most industrial control processes. PID stands for “Proportional, Integral, Derivative”, which identifies the three primary things that you can use to control errors (i.e. the gap between the actual demand and your current setting). You can code the algorithm to be proportional to the current error, or to the rate of change of that error, or to the sum of the errors over time.

Note that you get a transaction revenue value and the cost of a web-server, a Java-server and a Database server per turn, so these may also influence your code. For example – if the transaction revenue is very low relative to the infrastructure cost, then maybe it’s worth not having capacity for the peaks if it means you can economize on infrastructure for the majority of time. Or if the transaction revenue is very high, then it’s worth being much more aggressive with scaling up and slower scaling down, so you capture as much revenue as possible.

Don’t worry about getting that last 1% of efficiency until you have the whole application running.

## Discussion on Approaching this Project

Good communication and team-working is critical to a successful outcome. Spend time reading this document 2 or 3 times, and discussing each section with your team mates to see if you all have the same understanding. Then decide how you want to lay out your project plan for the project, taking into account the strengths and unique talents of your team.

You probably want to split the work into three pieces:

1. Have somebody code the GUI or the Web Server.
2. Have somebody work on the network programming part.
3. Have somebody work on the control algorithm and the overall linkage of the components.

Remember, the code needs to look pretty – i.e. well laid out, good choice of variable names, utilizing the features of your language to make elegant code, and lots of comments. (You also need to put a LGPL or GPL comment header onto your work, to explicitly mark it as Open Source.)

As you get more tired, make sure you are saving versions of your code on a regular basis, as you may well have to back-track to a previous point. Also, pace yourselves, and make sure you take time away from the challenge too, to rest, refuel and rehydrate!

Good luck! We hope that this is an enjoyable and intellectually challenging assignment for you!

© 2013, Morgan Stanley EI CTO Team