SGSN-MME Test Node Pool

Resource utilization for SGSN test nodes

ZANDRA NORMAN

Master’s Degree Project
Stockholm, Sweden January 2014

TRITA-ICT-EX-2013: 254
The SGSN-MME node, which is important in wireless networks, handles many users and therefore the uptime requirements for it are very high. The goal at Ericsson is to reach 99.9999% uptime for their SGSN-MME nodes and to reach this a lot of testing is required. Therefore the test process during the SGSN-MME development is both resource expensive and time consuming.

To optimize both resource utilization and test runtimes a common test node pool solution for their different test tools has been proposed. During this thesis a first exploratory investigation about how to optimize such a solution was made.

During the investigation different aspects were evaluated and a first input about how an optimal solution can be implemented is proposed. By having a scheduling layer in the common node pool, which determines how many nodes each regression job will get, depending on current load, the number of test cases in the job and the current node utilization optimized solutions can be found.

Future work in the area is still needed, but the exploratory research made during this thesis will give a good base to continue from.
Sammanfattning

Software Engineering of Distributed Systems
School of Information and Communication Technology

Masterprogram

SGSN-MME Test Node Pool

by Zandra Norman


Under undersökningen utvärderades olika aspekter och en första inblick om hur en optimal lösning kan implementeras föreslås. Genom att ha ett schemaläggnings-lager i den gemensamma nod-poolen, som fastställer hur många noder varje test-jobb får, beroende på den nuvarande lasten på poolen, antalet testfall i jobbet och den nuvarande nodutnyttjandet, kan en optimerad lösning hittas.

Framtida arbete inom området behövs fortfarande, men denna första undersökning ger en bas att bygga vidare på.
# Contents

## Abstract

## Sammanfattning

### 1 Introduction

1.1 Testing SGSN/MME ........................................... 2
1.2 Motivation/Problem ........................................ 2
1.3 Objective .................................................. 2
1.4 Delimitations .............................................. 2
1.5 Outline .................................................... 3

### 2 Background

2.1 Test tools and environment ................................. 4
   2.1.1 Test Tool 1 ........................................... 4
   2.1.2 Test Tool 2 ........................................... 5
   2.1.3 Test Tool 3 ........................................... 5
2.2 Related work and inspiration ............................... 5
   2.2.1 Managed Cloud Computing ............................. 5
   2.2.2 Hadoop YARN ........................................ 6
   2.2.3 Real-time MapReduce Scheduling ..................... 6

### 3 Test Node Pool Overview

3.1 Requirements ............................................. 7
3.2 Overview .................................................. 8
3.3 Resource Management ..................................... 9
3.4 Specialization Manager ................................... 10
3.5 Scheduling Layer ......................................... 10
3.6 Client/Interface .......................................... 10
3.7 Resource process ......................................... 10
3.8 Request Process .......................................... 11

### 4 Scheduling

4.1 Request Overview ......................................... 12
4.2 Different FIFO Strategies ................................. 13
4.3 Dependencies ............................................. 14
5 Evaluation 15
5.1 Method ................................................. 15
  5.1.1 Simulations ........................................ 15
  5.1.2 Measurements ..................................... 16
5.2 Investigation .......................................... 16
  5.2.1 Min/Max ............................................ 16
  5.2.2 Strategies ........................................ 17
  5.2.3 Number of nodes .................................. 17
  5.2.4 Load ............................................... 18
  5.2.5 Tool ............................................... 18
  5.2.6 Priority ........................................... 19
  5.2.7 Dynamic allocation ............................... 19
5.3 Results ................................................. 19
  5.3.1 Min/Max ............................................ 19
  5.3.2 Strategies ........................................ 22
  5.3.3 Number of nodes .................................. 26
  5.3.4 Load ............................................... 29
  5.3.5 Tool ............................................... 32
  5.3.6 Priority ........................................... 36
  5.3.7 Dynamic allocation ............................... 36

6 Conclusions 38
  6.1 Min/Max ............................................. 38
  6.2 Strategies ........................................... 39
  6.3 Number of Nodes ...................................... 39
  6.4 Tool .................................................. 39
  6.5 Priority .............................................. 40
  6.6 Dynamic allocation .................................. 40
  6.7 Final Conclusions ................................... 41

7 Future work 42
  7.1 Investigate Dynamicity further ...................... 42
  7.2 Return some nodes ................................... 42
  7.3 Allocating more nodes after start .................. 43
  7.4 Chunks instead of Slots ............................. 43

Bibliography 44
Chapter 1

Introduction

In recent years, the number of mobile devices that are able to wirelessly connect to the Internet has increased rapidly. Nowadays millions of people use the Internet through these mobile devices every day. This wireless Internet connection is made possible by the second, third and fourth generations (2G, 3G and 4G) of wireless networks. 4G is still young and growing, and through its all IP approach it promises even better services than its ancestors.

When connecting to the Internet through 2G or 3G, all traffic will pass through a node called SGSN (Serving GPRS Support Node)[1], which in 4G migrated to a node called MME (Mobility Management Entity)[2]. Ericsson has for a long time developed an SGSN node which is widely used, and when 4G came along they put the SGSN and MME functionality into the same node, calling it SGSN-MME[3]. This node makes it possible to serve all these three generations of wireless networks through one and the same node.

Each one of these SGSN-MME nodes is able to serve up to 18 million users, which means that if one node goes down up to 18 million users can lose connectivity. Therefore, the goal at Ericsson is to be able to guarantee 99.9999% uptime for their SGSN-MME nodes[4]. Considering their restart time which can take around 10 minutes, this means that if one node goes down one time during a year, at least 20 other nodes has to be up throughout the same year. To be able to provide this a lot of testing is required, so the testing process during the SGSN-MME development is very important and time consuming.
1.1 Testing SGSN/MME

During the testing process in the SGSN-MME development some different regression test tools, test nodes, and test runners are used. Currently three different regression test tools are used, all handling node installation and test case distribution. Test runners are used to simulate the environment for each test node. A short explanation of the different regression test environments follows in chapter 2.

1.2 Motivation/Problem

For each of the test runners a fixed number of virtual machines and servers are available. Today they also use different kinds of nodes and has their own node pools. In the near future though the plan is to use the same kind of nodes for some of the different test runners. In this new situation it might be useful to also use a common node pool which can maximize the utilization of the resources and through this be able to improve test runtimes. With a common node pool it is important to give each job as many nodes as needed without draining the resources and starving other tools.

1.3 Objective

The aim of this thesis is to examine how to improve machine utilization through a common node pool, and through that be able to optimize the regression test runtimes. To make this possible an optimal scheduling strategy for the allocation of the shared resources is needed. The main question addressed during the thesis is:

How can the resources be shared to optimize machine utilization, and the regression runtimes with a common test node pool?

1.4 Delimitations

This is only an exploratory study to determine the optimization possibilities with a common resource pool for the SGSN-MME test nodes. The focus is to investigate if a common node pool can be valuable and if investigations around it should continue. Two of the three test tools will share the same resources, so the focus of the evaluation during the project lies in how to share the resources between these two. Support to add the third tool is implemented, but since it will not interfere with the other tools during the
scheduling it is not evaluated during the project. The evaluation is made in a general way though, so it will be useful for when adding support for the third tool as well.

1.5 Outline

The report is organized as follows. In the next chapter, chapter 2, an extended background and related work is provided. Then, in chapter 3, the system structure for the Test Node Pool is described. Chapter 4 presents the different scheduling strategies and constants evaluated, followed by the evaluation and results in chapter 5. Conclusions from this evaluation is provided in chapter 6 and finally, in chapter 7, future work is listed.
Chapter 2

Background

In this chapter a brief explanation of the different test tools, test runners and test nodes are provided.

2.1 Test tools and environment

During all software development testing is an important part, and the goal for SGSN-MME, to guarantee 99.9999% uptime, makes it extra important during its development. Regression tests are run frequently to make sure that modification of the product does not have any unintended side effects. Below a short explanation of the different regression test environments follows.

2.1.1 Test Tool 1

The first test tool for SGSN-MME regression tests, called Test Tool 1 throughout this report, is used for test cases written in erlang. It runs on the first version of the SGSN node simulation. The plan is to migrate Test Tool 1 to be able to use the next generation of simulated nodes which are more effective, and therefore this is what was investigated during the thesis.

There is a test distribution tool that is used to distribute the Test Tool 1 test cases, here called Distribution Tool 1. Distribution Tool 1 can run different jobs in parallel, but if a job triggered cannot be handled immediately it is queued until enough machines, and through that simulated nodes, are available.
2.1.2 Test Tool 2

The next test tool, here called Test Tool 2, is a new test tool that can run test cases which earlier only could be run on the nodes used in Test Tool 3, in a simulated environment. It uses the next generation of simulated nodes.

The test distribution tool used for Test Tool 1, called Distribution Tool 2 in the report, can currently only run one job at a time and the rest are queued to wait for its turn.

2.1.3 Test Tool 3

In the final test tool, Test Tool 3, test runner a completely different version of nodes, requiring other machines than the above ones. This will later on be included in the Test Node Pool investigated during the thesis, but since it will have its own quotas it is not investigated in the report. The common solution is wanted to make the management and maintenance easier in the future. Test Distribution Tool 3 can like Test Distribution Tool 2 only run one job at a time and the rest are queued in FIFO fashion.

2.2 Related work and inspiration

The system implemented is not just a simple node pool. It also handles scheduling for maximal utilization and specialization of the nodes. This has been inspired from some different already well known technologies presented below.

2.2.1 Managed Cloud Computing

The pool need to be able to prepare and specialize the nodes for the different incoming requests. When looking at Amazon EC2[5] and OpenStack[6] they use server images as a template which it is possible to launch instances from. This can be used as inspiration for the specialization layer in the Test Node Pool. In Amazon EC2 instances launched run until they are stopped or they fail.
2.2.2 Hadoop YARN

In Hadoop YARN[7], a resource management platform, each node has its own manager process keeping track on information about the node and keeping contact to the centralized resource manager. This inspired the Test Node Pool solution with a centralized resource manager and distributed resource processes at each resource. The process on the resource side can be extended to monitor the node usage later on if needed. The resource manager in Hadoop-YARN has a scheduler which schedules the requests from different competing applications and it allocates nodes based on this scheduling.

2.2.3 Real-time MapReduce Scheduling

In the real-time MapReduce Scheduling solution presented in [8], a constraint based scheduling algorithm is suggested. This algorithm was implemented in the constraint programming platform Gecode[9], which could also be a good tool to use for other scheduling problems like the one during this thesis. Gecode requires some background knowledge though and is not elementary to understand and start using, and since one requirement on the Test Node Pool is that it should be easy to maintain, this thought was put aside for this thesis investigation.
Chapter 3

Test Node Pool Overview

The name of the pool implemented during this thesis, Test Node Pool, comes from the final goal to be able to handle both simulated nodes and another version of test nodes. The pool is based on a number of goals and requirements presented below, and then an implementation overview is provided.

3.1 Requirements

There are some different goals with and requirements on the Test Node Pool. These goals and requirements are listed below.

• Optimize resource utilization (I)
  
  – The main goal with the common pool is to improve the resource utilization. This can be achieved by sharing the resources among the different tools, so that if the load on one tool is low and another tool with high load, the tool with higher load should be able to use more nodes than if the different tools had fixed quotas. With fixed quotas the resources for a tool with low load will be unused even if another tool has high load and is in need for more resources.

  – It is also important to use as many nodes as possible as much as possible. This means that the pool needs to allocate as many nodes as possible for each regression job without draining the pool and delaying other jobs. The allocation should be done in a fair way so that all jobs and tools get optimal allocations.

• Optimize regression runtimes (II)


Both the total runtime for all jobs and the runtime for single jobs are important to take into account. With an optimal total runtime there might still be single jobs suffering from the optimizations which should be avoided.

- Scalable (III)
  - The pool should be able to scale both in respect of number of available resources and the load on the pool, i.e. the number of regression jobs that are requesting nodes.

- Easy to maintain (IV)
  - An important aspect is that the Test Node Pool should be easy to maintain. It is easier to maintain one pool than to maintain many, and therefore there is a wish for adding the test nodes used by Test Tool 3 to the node pool.

- High availability (V)
  - If the pool goes down no regression jobs will be able to run and the testing will be freezed. Since the test process during the SGSN-MME development already requires a lot of time extra delay through this is not acceptable.

- Specialization support (VI)
  - Different tools require different preparations and specializations of the nodes. When a node is delivered to a tool it should be ready, so that the tool can start the regressions as soon as possible.

- Priority possibility (VII)
  - When a team is close to a release there might be extra important that the regression tests finish as fast as possible. Therefore priority possibilities was requested.

### 3.2 Overview

To reach these goals and fulfill the requirements a layered architecture was chosen which makes it easy to maintain (IV). It contains four layers; a resource manager, a specialization manager, a scheduler and an interface. These layers, together with a process running on each of the resources (test node machines) provided, provide all functionality needed for the pool. In this section these different parts are briefly described and an overview is provided in figure 3.1.
3.3 Resource Management

The lowest layer in the pool is a resource manager. Its one and only task is to keep track of the different resources connected to the pool and their state. From this resource manager above layers can allocate resources, and later free them when they are not needed any more. It also keeps track of the state of each resource, making it possible to list information about what are currently in use, which jobs are running, how many resources are available and so on.

To provide high availability (V) without losing data it is possible to run more than one resource manager in a cluster. These different resource managers synchronize all information upon any change, so if one goes down the others will still have the right information. If only one resource manager is run it will be restarted on crash and recover to its last state with help from a persistent database.
3.4 Specialization Manager

The specialization manager is the next layer in the pool, and it handles installation and cleaning of nodes. When all nodes from an installation request are installed they are returned to the caller with help from a callback function.

This layer was inspired by the machine image solutions in Amazon E2 and OpenStack where virtual machines are created with help from pre-configured operating systems and virtual application software. Since the specializations needed were not specified the actual specialization is currently not implemented yet, but the flow and skeleton code is written to make it easy to add the actual specializations later on in the development.

3.5 Scheduling Layer

The scheduling layer handles the other requirements, (I), (II), (III), (VII). It provides functionality to make sure that the resources are utilized as well as possible without starving any of the requests. It schedules each request and decides how many nodes the request will be assigned. If there are currently not enough resources to meet a request it is queued.

Since the focus in this thesis is to utilize the resources as well as possible and optimize the regression runtimes some different scheduling strategies were evaluated. These strategies are presented in the next chapter.

3.6 Client/Interface

The top layer is simply an interface to the outside world. This is implemented as a simple REST interface which is implemented with help from Cowboy[10], which is a small, open source, fast and modular HTTP server.

3.7 Resource process

Each machine connected to the pool, i.e. each pool resource, is running a process which handles all communication with the pool functionality. When this process starts it connects to the pool’s resource manager, and when it is stopped it disconnects from it. It also handles installation and cleaning of the node when required from the specialization
manager, and startup on demand from the client layer.

In the future it might be needed to keep track on what happens on the resource side, if the nodes allocated are used or if they are idle and so on. In that case this is the place to add this functionality, since the process is already running on the right side. If a node is idle for too long it might be useful to return the node to the pool, even if the tool or user that has allocated it does not handle this freeing by itself.

3.8 Request Process

When a regression test tool has a job that needs to be run it sends a scheduling request to the Test Node Pool. The interface layer receives the request and forwards it to the scheduler layer. When the scheduler layer has decided how many nodes the job will get and it is time to run the job it sends an installation request to the specialization layer. Then the specialization layer allocates enough nodes from the resource manager layer and prepares the nodes for their usage. When all nodes are installed and ready they are returned to the scheduling layer who keeps track of them and waits for a start request from the interface. This start request is sent when the test tool which had the initial scheduling request sends a start request to the pool. If the nodes are not ready yet it will have to resent the start request later, in a polling fashion. If the nodes are ready they will be returned to the test tool which can start running the regression job.

When the job is finished the test tool returns the nodes to the pool, and when they have been cleaned in the specialization manager the nodes are freed to the resource manager and can be used for a new job.
Chapter 4

Scheduling

The goal of this thesis is to investigate how to utilize the resources as efficiently as possible and see what effect that can have on the different regression runtimes. To reach an as good utilization as possible the scheduling layer is very important. Therefore some different scheduling strategies is tested and evaluated during the thesis work. The different strategies are described in this section/chapter. The strategies have some core functionality in common but differ in how they schedule the incoming requests. In this thesis four different strategies were investigated; a simple FIFO strategy, a FIFO strategy for maximizing number of concurrent jobs, a soft deadline strategy without preparing notifications and a soft deadline strategy with preparing notifications.

4.1 Request Overview

The different strategies all provide the following functions:

- **Schedule**: scheduling the request; when it will be handled and how many nodes it will be assigned. This is where the focus of the evaluation lies, and where the different strategies differ the most.

- **Cancel**: can be called if a scheduled request is not needed anymore. Then it is removed from the queue and if the nodes are already allocated and installed they will be cleaned and returned by and to the lower layers.

- **Start**: when the requester wants to start using the node(s) it has sent a schedule request for, it calls the start function. If the scheduler has already gotten a list of installed nodes for this specific job they are returned to the requester, otherwise an estimated delay time is sent to the requester who can call again when this time has passed.
Finished: when a tool/user is finished with its node(s) they are returned by calling the finished function. Then they are cleaned and returned by and to the lower layers.

Since also checkout requests are handled in the system, and these checkout requests should be met right away, a separate queue is created for these requests. They are always handled before any new regression request can be handled. The scheduler also makes sure that the regression request never leaves less than CO_RESERVED nodes left in the pool. The checkout requests are always handled in FIFO order since they are viewed as equally important and all of them only require one node.

### 4.2 Different FIFO Strategies

The base for all strategies used is a simple FIFO queue. It puts new request at the back of the queue and handles the first job in the queue as soon as it can.

The first strategy only takes into account the number of test cases for the request’s regression job and possibly the tool the request comes from and calculates the minimum number of nodes required to start the regression (min) and the maximum number of nodes this regression will be allowed (max) to allocate from this information. It allocates as many nodes as possible so if more than max nodes are free max nodes are allocated. If less than min nodes are available the job has to wait until enough nodes are freed. If the number of available nodes is something in between the min and max all nodes are allocated for the job.

The second one works in the same way but also takes the current queue into account trying to fit in as many jobs as possible every time a new job can be run.

To enable prioritization of jobs, a simple real time priority queue is used instead. If a job is added as usual, without any higher priority it will be added in the end of the queue by adding X to the time stamp when added. If the job has a higher priority Y will be added instead, and since the queue is sorted by these stamps the job with the higher priority will get a better position in the queue. By using the adding time like this the risk of starvation is eliminated, but the value of X and Y has to be wisely chosen to make sure that it actually has an effect without having too big of an effect, making non-prioritized jobs wait for too long.
4.3 Dependencies

To decide how many nodes each job should have some different constants are used. The number of nodes are depending on the number of test cases for the job and which test tool makes the request. The number of test cases should have impact since a job with more test cases needs more nodes than one with fewer to be able to run within a reasonable time. Since the average runtime for the test cases differs for different test tools the test tool should also be considered when deciding the number of nodes for a job. The total number of available nodes and the current load on the system should also have an impact in this allocation calculation.

A minimum and maximum number of nodes are used to make sure that each job will get enough nodes to run without draining the pool.
Chapter 5

Evaluation

During this thesis an investigation about how to improve resource utilization and regression runtimes through a common node pool for the SGSN-MME development test process was made. To achieve optimal resource utilization an efficient scheduling strategy is needed. The scheduling was therefore evaluated in respect to different strategies and constants, and this evaluation is presented below.

5.1 Method

The evaluation was made through simulation runs with different conditions. Data was extracted from the simulation runs and measurements were made as explained below.

5.1.1 Simulations

Since the investigation during this thesis is exploratory and the Test NodeF Pool is still under construction it is not ready for real system experiments. Furthermore, real system experiments would require too much time. Therefore a simulator was implemented, based on data from previous regression runs from the different test tools. An event queue and logical time was used to simulate the real system and the different regression job requests. The runtime for each job was calculated based on the previous data and the number of nodes the job got. By adding and removing jobs the load could be changed, simulations with different amount of nodes could be run, and by changing constants in the scheduling algorithm different calculations for the allocations could be tested.
5.1.2 Measurements

The results from the simulations were compared based on some different aspects. To get an optimal resource utilization it is important to look at the total runtime for all jobs in the simulation, how the resources are used, the queueing required before getting to start the jobs and the runtime for specific jobs.

The total runtime is important to optimize since time saving is always important in any development process. It shows how well the resources are used, not only if they are used or not. The runtime for each job is measured from the time the job was added, i.e. when the scheduling request was made, to when it is finished, i.e. when the nodes are returned to the Test Node Pool. The sum of all these runtimes gave the total runtime. The runtimes are presented in graphs.

The resource utilization measurement shows if the resources are used or not. Together with the total runtime it can show if the resources are well utilized or not, not only if they are allocated for a job but also if an appropriate amount of nodes are allocated for the jobs. If too few nodes are allocated for a job the time to run it will be longer, but if too many nodes are allocated the resources in the pool will be drained. With a drained pool jobs have to queue which requires more time that necessary. Both the resource utilization and job queueing are visualized in graphs.

Even if the total runtime is fine there is a risk for single jobs to suffer if the scheduling is not optimized properly. Since this should also be avoided, and a fair and efficient scheduling and allocation strategy is wanted, it is important to look at single jobs and how they suffer. In the report this is presented through tables which shows how many jobs are suffering for different simulation runs.

5.2 Investigation

Optimal constants, calculation and scheduling strategy are needed to optimize the utilization and regression runtimes. To find this some aspects has to be taken into account and investigated. These aspects are presented in below.

5.2.1 Min/Max

To determine how many nodes a job should get the scheduling layer uses the number of test cases and performs a calculation from this. A job has a maximum number of nodes
it can get and a minimum number of nodes required to start it. The min and max are based on how many test cases the job has. A job requires min nodes per X test cases and can get max*min nodes per X test cases. In other words, max is a constant deciding the range of the number of nodes allowed for each job. Both the size of these constants and the relationship between them was investigated.

The values evaluated for min were the following: 1, 2 and 4 nodes per X test cases. By setting max to 1 the minimum and maximum allowed nodes would be the same, i.e. the range is 0. To see what effect the range had simulations with max set to 1, 2, 4, 8, 16 and 32 were run for each min value. Here it was important to look at both the total runtime and resource utilization, job queueing and the runtimes for specific jobs to make sure that jobs were not suffering too much. The tests are run with 400 nodes and with normal load.

5.2.2 Strategies

Simulations with the different strategies described in chapter 4 were run. By comparing the runtimes, resource utilization and queueing effect from the different strategies the best one could be chosen for further use.

5.2.3 Number of nodes

The amount of connected resources and through that, the number of available nodes, might change over time. Sometimes servers can be down or occupied with other work, and later on more resources might be added or removed. The varying amount of resources require that the pool adapts the node allocations to fit the current state as well as possible. If nodes are added more nodes should be allocated for each job to utilize the resources in an efficient way. If nodes are removed less nodes should be allocated to make sure that the resources are not drained, causing too long job queues. To investigate how to adapt simulation tests were run with different amount of available resources.

Simulations for 200, 400 and 600 available nodes were run and presented in the report. For these runs the total runtimes, resource utilization and job queueing are important to look at. The min and max constants are changed in the same fashion as described in previous section and the load is kept on normal level.
5.2.4 Load

The load on the system also varies over time. Therefore it is important to investigate what effect the load has on regression test runtimes and resource utilization and how to adapt to varying load.

This was investigated by running simulations with different amount of jobs during a fixed time. The min and max section already tested how the normal load affected. To investigate other loads first half the jobs were randomly removed from the normal load and simulations ran with the new job list showing the effect from a half load run. Then the number of jobs in the normal load run were doubled, to run the simulations with double load. During all these simulations the number of nodes were set to 400 and the min and max values varied as described in section 5.2.1.

As before it was important to measure the total runtime, the resource utilization and the queueing to evaluate the different loads on the system. This to see how and why adaptations should be made if the load varies.

5.2.5 Tool

The average runtime for the test cases that runs on the different test tools differs and the demands on the different test tools might be different in the future. Therefore the number of nodes could be adapted depending on which tool makes the request. The test cases ran on Test Tool 1 are slower than the ones running on Test Tool 2 and therefore an investigation about how the relationship between these could affect both the total runtime, and the total runtimes for each tool. The exact time demands are not specified yet, but during this thesis an investigation how to adapt to time requirements later on was made.

When evaluating this a relationship between the Test Tool 2 requests and the Test Tool 1 requests is needed. Tests when they are the same were run as described in section 5.2.1, then Test Tool 1 got 3 times as many nodes per X test cases as Test Tool 2, and finally 7 times as many nodes. During these runs the number of available nodes in the pool were 400 and the load was normal.
5.2.6 Priority

One of the requests on the pool were priority support to be able to run extra important regression jobs faster than others. The simple real time scheduling solution described in chapter 4 was implemented to make this possible.

Here, the jobs are scheduled by adding some seconds to the current time (in seconds) when the allocation request was made. How many seconds that should be added was decided with help from a constant, so an evaluation about what to set this constant to if a job was prioritized or not was made. The difference between a prioritized and an ordinary job was set to 500s, 1500s, 2500s, 5000s and 10000s.

This is tested by prioritizing one job and comparing its start time and finish time for the different simulation runs. Then two jobs close to each other in the job event queue, were prioritized to see if they canceled each other or a similar effect was shown. The simulations were run with 400 nodes and normal load.

5.2.7 Dynamic allocation

When the above had been investigated a very simple dynamic allocation solution was tested to show how it could work in the future. This should be further investigated later on, but gives a first idea about how it can be implemented.

5.3 Results

The results from the simulations are presented through graphs and discussed in this section.

5.3.1 Min/Max

The results from these experiments are presented in figure 5.1, 5.2 and 5.3 and table 5.1. Figure 5.1 shows the runtimes, figure 5.2 visualizes the resource utilization over time, figure 5.3 the queueing over time. Table 5.1 presents the relationship between single jobs ran with the different ranges for the optimal max value.

In figure 5.1 it gets clear that the max value is very important to optimize the total regression runtime. The best runtimes are reached when
Figure 5.1: The runtime for min and max constant investigation simulations. "tc" stands for test cases.

- min is 4 and max is 8*min (I)
- min is 2 and max is 16*min (II)
- min is 1 and max is 32*min (III)

When looking at these values it gets visible that the actual value of the max allowed number of nodes per X test cases in all these cases are 16. Since this is what will be allocated for each job unless jobs are queued or the resources are drained, it has a large impact on the runs. When looking at the resource utilization in figure 5.2, and the job queueing in figure 5.3 together it gets clear what impact the resource utilization has as long as queueing is avoided. If too few nodes are allocated for the different jobs they will naturally run slower, but the queueing also has a large impact on the total runtime, so as soon as too many jobs has to queue before getting to start the total runtime will be affected.

In table 5.1 the relationship between single jobs runtimes is presented. The runs chosen for this are the best ones presented above; (I), (II), (III). Even if the total runtimes are very similar, with only a few percents differences there are some jobs that are clearly suffering. When comparing (I) to (II) 44% of the jobs in (II) are more than 10% faster than the ones in (I), but only 10% of the jobs in (I) are more than 10% faster in (II). When comparing (I) and (III) a similar relationship is shown. This shows that it is important to start jobs as soon as possible, even if it is started with fewer jobs than if it
Figure 5.2: Resource utilization for min and max constant investigation simulations.

(a) 1 node per X test cases
(b) 2 nodes per X test cases
(c) 4 nodes per X test cases

Figure 5.3: Job queueing for min and max constant investigation simulations.

(a) 1 node per X test cases
(b) 2 nodes per X test cases
(c) 4 nodes per X test cases
was queueing before. A larger range makes it possible to start the jobs with fewer nodes and through this it decreases the queueing time for the jobs.

Table 5.1: Single jobs suffering comparison, max is 16 nodes per X test cases.

<table>
<thead>
<tr>
<th>I: min is 4 nodes per X test cases</th>
<th>Percent better</th>
<th>0%</th>
<th>10%</th>
</tr>
</thead>
<tbody>
<tr>
<td>II: min is 2 nodes per X test cases</td>
<td></td>
<td>28%</td>
<td>10%</td>
</tr>
<tr>
<td>III: min is 1 nodes per X test cases</td>
<td></td>
<td>56%</td>
<td>44%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I: min is 4 nodes per X test cases</th>
<th>Percent better</th>
<th>0%</th>
<th>10%</th>
</tr>
</thead>
<tbody>
<tr>
<td>II: min is 2 nodes per X test cases</td>
<td></td>
<td>40%</td>
<td>11%</td>
</tr>
<tr>
<td>III: min is 1 nodes per X test cases</td>
<td></td>
<td>54%</td>
<td>47%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I: min is 4 nodes per X test cases</th>
<th>Percent better</th>
<th>0%</th>
<th>10%</th>
</tr>
</thead>
<tbody>
<tr>
<td>II: min is 2 nodes per X test cases</td>
<td></td>
<td>36%</td>
<td>12%</td>
</tr>
<tr>
<td>III: min is 1 nodes per X test cases</td>
<td></td>
<td>28%</td>
<td>19%</td>
</tr>
</tbody>
</table>

5.3.2 Strategies

The two different strategies evaluated had similar runtime results as visible in figure 5.4, but with some important differences. The first and simplest strategy was a bit more efficient in the most optimized allocations but a bit more unstable when the allocation number was off from the optimal. The second strategy was more stable for a larger range of allocation constants.

At a first glance the node utilization and job queueing, shown in figure 5.5, 5.6, 5.7 and 5.8, for the different strategies also look very similar. After a closer look though, it gets clear why Strategy 1 is more unstable for bad values than Strategy 2. For Strategy 1 the job queueing is worse for bad values which results in worse runtimes.
Figure 5.4: Runtime results for strategy investigation simulations.

(a) 1 node per X test cases

(b) 2 nodes per X test cases

(c) 4 nodes per X test cases
Figure 5.5: Resource Utilization for strategy investigation simulations; Strategy 1.

Figure 5.6: Resource Utilization for strategy investigation simulations; Strategy 2.
Figure 5.7: Job queueing for strategy investigation simulations; Strategy 1.

(a) 1 node per test cases
(b) 2 nodes per X test cases
(c) 4 nodes per X test cases

Figure 5.8: Job queueing for strategy investigation simulations; Strategy 2.

(a) 1 node per X test cases
(b) 2 nodes per X test cases
(c) 4 nodes per X test cases
5.3.3 Number of nodes

In this section results from the different number of nodes available in the system are presented in figure 5.9 to 5.13. Figure 5.9 shows the total runtime for 200 and 600 available nodes, figure 5.10 and 5.11 shows resource utilization and figure 5.12 and 5.13 the queueing. To see this information from simulation runs with 400 available nodes, view the previous section.

![Figure 5.9: Runtime for investigation simulations with different number of nodes available.](image)

In figure 5.9 a similar relationship for min and max as in section 5.3.1 is visible here, but for 200 nodes the runtime is optimal when the maximum is 3 nodes per X test cases.

For the runs with 600 nodes it is optimal when the maximum is 16 nodes per X test cases, but when comparing the runtime graph here with the one in figure C it is visible that the graph here has a small offset towards a larger value. This shows that the maximum allowed nodes is slowly increasing when the number of nodes available in the pool is increasing.

When looking at the resource utilization and queueing, presented in figure 5.10, 5.11, 5.12 and 5.13, the fashion is also similar to the previous section. Bad resource utilization results in bad runtimes and draining the resources has the same effect. A balance between the two is needed, but how to find it is not obvious since the balance is found on different points for the different simulations run here. The balance has to be found depending on the number of nodes available in the pool.
Figure 5.10: Resource utilization for investigation simulations with different number of nodes available; 200 nodes.

Figure 5.11: Resource utilization for investigation simulations with different number of nodes available; 600 nodes.
Figure 5.12: Queueing for investigation simulations with different number of nodes available; 200 nodes.

Figure 5.13: Queueing for investigation simulations with different number of nodes available; 600 nodes.
5.3.4 Load

The results from differing load is very similar to when changing the number of nodes, as shown in figure 5.14 to 5.18. By changing the load the balance between load and number of nodes available chances in a similar way as when changing number of available nodes, and hence these results are expected.

![Figures 5.14](image)

(a) Half Load

(b) Double Load

**Figure 5.14:** Runtime for load investigation simulations.

The load is not known in advance though, and since it cannot be measured directly other measurements are needed to estimate the load during runtime. When looking at figure 5.15, 5.16, 5.17 and 5.18 it gets visible that the load on the system can be estimated by measuring the length of the queue and how well the resources are utilized. If many nodes are unused, the utilization is bad and the number of nodes allocated for each job should be increased, and if the queue is long the resources are drained and it should be decreased.

By changing the load the balance between load and available nodes chances in a similar way as when changing number of available nodes, and hence these results are expected.
Figure 5.15: Resource utilization for load investigation simulations; Half Load.

Figure 5.16: Resource utilization for load investigation simulations; Double Load.
Figure 5.17: Job queueing for load investigation simulations; Half Load.

Figure 5.18: Job queueing for load investigation simulations; Double Load.
### 5.3.5 Tool

In figure 5.19 to 5.25 the results from the tool balancing are shown.

![Figure 5.19](image)

(a) Total runtime for jobs from both tools

![Figure 5.20](image)

(b) Total runtime for Test Tool 2 jobs

![Figure 5.21](image)

(c) Total runtime for Test Tool 1 jobs

**Figure 5.19:** Runtimes for tool dependency investigation simulations.

As visible the total runtime is more impacted on the Test Tool 1 runtime changes than on the Test Tool 2 runtime changes, which is natural since Test Tool 1 has more jobs and longer runtimes. It is still important to keep the right balance in amount of allocated nodes, so if the relationship between Test Tool 2 and Test Tool 1 is 1 to 3 the min and max constants should be decreased a bit. For relationship of 1 to 7 these constants should be decreased even more as visible in figure 5.19.

Figure 5.20, 5.21, 5.22, 5.23, 5.24 and 5.25 shows a similar relationship between the node utilization and job queueing as the above simulations. A balancing solution where the nodes are well utilized without causing too much job queueing gives the optimal runtimes.
Figure 5.20: Resource utilization for tool dependency investigation simulation; Original.

Figure 5.21: Resource utilization for tool dependency investigation simulation; Times 3.
Figure 5.22: Resource utilization for tool dependency investigation simulation; Times 7.

Figure 5.23: Job queueing for tool dependency investigation simulations; Original.
Figure 5.24: Job queueing for tool dependency investigation simulations; Times 3.

(a) 1 node per X test cases
(b) 2 nodes per X test cases
(c) 4 nodes per X test cases

Figure 5.25: Job queueing for tool dependency investigation simulations; Times 7.

(a) 1 node per X test cases
(b) 2 nodes per X test cases
(c) 4 nodes per X test cases
5.3.6 Priority

The results, presented in table 5.2, shows that it works as planned without having a big impact on the total runtime as long as the time constant is set to appropriate values. The jobs not prioritized are of course suffering some but that is unavoidable. They do not suffer in any unreasonable way and are not starved. A problem with the algorithm is showed in table, where the runtime becomes longer, for example for (A1, 1500), where even if the job starts earlier it takes longer since it gets fewer nodes. This is a risk in this kind of scheduling, and it should be investigated further how to handle this situation.

Table 5.2: Comparison with the original queueing time and runtime for the prioritized jobs.

A1 is the first situation, with only one prioritized job.
A2 and B2 is the second situation, with two prioritized jobs.
The run time is from the time the job was added until it finishes.

<table>
<thead>
<tr>
<th>Time constant</th>
<th>500</th>
<th>1500</th>
<th>2500</th>
<th>5000</th>
<th>10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1: queuing time</td>
<td>-21%</td>
<td>-23%</td>
<td>-23%</td>
<td>-23%</td>
<td>-23%</td>
</tr>
<tr>
<td>A1: run time</td>
<td>-26%</td>
<td>+58%</td>
<td>+58%</td>
<td>-32%</td>
<td>-32%</td>
</tr>
<tr>
<td>A2: queuing time</td>
<td>-21%</td>
<td>-23%</td>
<td>-23%</td>
<td>-23%</td>
<td>-23%</td>
</tr>
<tr>
<td>A2: finish time</td>
<td>-26%</td>
<td>+58%</td>
<td>+58%</td>
<td>-32%</td>
<td>-32%</td>
</tr>
<tr>
<td>B2: queuing time</td>
<td>-24%</td>
<td>+34%</td>
<td>-24%</td>
<td>-24%</td>
<td>-24%</td>
</tr>
<tr>
<td>B2: finish time</td>
<td>-48%</td>
<td>-26%</td>
<td>-35%</td>
<td>-35%</td>
<td>-35%</td>
</tr>
</tbody>
</table>

5.3.7 Dynamic allocation

In figure X the runtime results from the dynamic allocation constant simulations are visualized. In figure X the runtime is compared with runtimes with different fixed min and max constant values and it gets clear that this dynamic solution is not optimal.
Figure 5.26: Runtime for dynamic allocation investigation.
Chapter 6

Conclusions

From the simulation results presented in chapter 5, some conclusions can be made. These conclusions are presented in this chapter, first separated by each experimental focus, and then, in the end of the chapter, together to make some common recommendations.

6.1 Min/Max

In section 5.3.1 it got clear that the maximum number of allowed nodes for a job has a large impact on the total runtime. This is natural since it is the number of nodes allocated most often as long as the jobs do not have to queue too much. It is therefore important to have an optimal value for the maximum allowed number of nodes, which in this particular case was 16 nodes per X test cases.

Another thing that is clearly also important is to balance the utilization and the job queueing. This was expected, since both bad utilization and a lot of queueing leads to bad runtimes. The queueing had a bit larger impact on the total runtime than expected though, and should be avoided as much as possible.

When looking at the runtimes for single jobs the same pattern got visible. By starting a job earlier and with a bit fewer nodes than it might wish for in an optimal case, it will most often finish faster than if it has to wait. Therefore, the range between the minimum number of nodes accepted to start a job and the maximum number of nodes allowed for the job should be relatively large.
6.2 Strategies

Two strategies for the queueing were tested, one where the first job in the queue always got as many nodes as possible, and one where as many jobs as possible shared the remaining nodes to be able to start them faster. In both situations the queue kept its FIFO order to make sure no jobs were starved.

It turned out that the second solution gave more stable low results even if the first solution was a bit faster in some cases. The reason for the more stable runtimes was the fact that as many jobs as possible are starting instead of the first job(s) with as many nodes as possible. Since a solution which will adapt to different situations is wanted and the second strategy is recommended.

6.3 Number of Nodes

When varying the number of nodes or the load similar results were shown since they have the same effect in the end. When the load is high compared to the number of available nodes, fewer nodes should be allocated for each job, to avoid draining the resources resulting in queueing jobs. When the load is low and compared to the number of nodes, more nodes should be allocated to utilize the resources in a good way. The difference between high load/few available resources and low load/many available resources was not as large as expected though. With a relatively high range between the minimum number of nodes required and the maximum number of nodes allowed the the pool could handle the differing load in an efficient way.

To adapt to this the load needs to be measured somehow. This can be done by keeping track of the job queue and the node utilization. If the queue is too long the pool is overloaded and the constants for min and max should be decreased, and if there are too many node slots unused, i.e. the resources are not well utilized, the min and max should be increased.

6.4 Tool

The results shows how to adapt the number of nodes allocated for a job to the tool requesting the allocation. It does not have any significant impact on the total runtime, so if this kind of adaption is wanted later it is safe and easy to add to the implementation.
If different requirements are put on the different tools in the future they can be adapted later on. Important to think about if this is done is that the same amount of nodes should be requested in total, to keep avoiding unnecessary queueing or bad node utilization. If the base calculation for min and max for example is:

- min = 4 nodes for X test cases
- max = 16 nodes for X test cases

which is optimal when all tools use the same calculation, but Test Tool 1 gets three times as many nodes as this base calculation the total amount of nodes allocated will of course be a lot more than optimal.

Since the requirements specifications are not clear yet the details for this were not investigated, but it is clear that it is possible to adapt to requirements later on if needed.

### 6.5 Priority

The priority possibilities were just briefly tested but showed that it is possible to have this without destroying the rest of the flow. The solution keeps the same FIFO order for the jobs except if a job has higher priority, and the real time element keeps makes sure that jobs will not starve. The problem showed in the result section, where a prioritized job risks getting fewer nodes resulting in longer runtime in total, should be investigated further.

### 6.6 Dynamic allocation

As visible from the load and number of available node slots simulations it is good if the constants for the calculations of minimum required nodes and maximum allowed nodes are adjusted if needed. Unfortunately the time was not enough to determine the best ways to do this, but the tests at least shows that the constants easily can be adjusted by measuring the job queue and resource utilization. In the tests ran during this project the adjustments were made too fast and therefore balancing values for the constants were never found. It gives a base to continue from later on.
6.7 Final Conclusions

The question to be answered during the thesis was as follows:

*How can the resources be shared to optimize machine utilization, and the regression runtimes with a common test node pool?*

This can be done by having a scheduling layer in the pool which determines how many jobs each job will get through a calculation. By making sure that the calculation gives a relatively large range of possible number of nodes and if the pool tries to start as many jobs as possible in case of job queues different loads can be handled to a certain level. This might be efficient enough if the system keeps a somewhat stable level on load and number of connected resources. If too large changes from these are rare and slow it might be more efficient to have a manual change of the min and max constants if needed then having to keep track on the adjustments in the pool.

If there often are too large changes in load or amount of connected resources and this cannot be handled anymore the constants determining the min and max for the node allocation range should be adjusted to fit the new situations as explained above. To be able to do so more investigation is needed.
Chapter 7

Future work

This thesis was a first investigation about the possibilities to use a common node pool for different test tools during the SGSN-MME development test processes. It gives an initial overview of how it can work and be optimized to improve the resource utilizations and runtime, but further investigation is needed. These future work areas are presented in this chapter.

7.1 Investigate Dynamicity further

During the thesis project a balanced dynamic solution was not properly investigated or found due to time constraints. More investigation and simulations in this area are needed but the experiments run here gives a base to continue from.

7.2 Return some nodes

If a job gets to start with fewer nodes that it prefers there is currently no functionality that makes it possible for the tool to request more nodes for the job after a while. If another job releases its nodes and the job queue is empty these nodes will be unused even if the first job would like to have some more nodes. It is also possible that even if jobs are queueing it might still be preferable to allocate some more nodes to the first job. Therefore, investigation in this area would be useful for a future common test node pool.
7.3 Allocating more nodes after start

In some cases a job might want to return some nodes that it no longer has use for. The current Test Node Pool does not handle this case and it is only possible to return all nodes at the same time. To handle this special case functionality for it should be implemented and investigated to see what effect it would have for the system.

7.4 Chunks instead of Slots

The different test nodes used in the testing of SGSN-MME are of different sizes. Today the Test Node Pool acts like all nodes are of the same size and each resource slot is big enough to fit in the largest node. This means that space is wasted if a smaller node is wanted, since it still needs to allocate a whole slot to be able to run. Thus can be solved by dividing the resources up into smaller chunks and then allocating just enough chunks to be able to run the node wanted so that the resources could be utilized even better. If the smallest node only requires one chunk and largest one requires three chunks, then three of the smallest nodes could be run on the same space that only one has to allocate in the current version.
Bibliography


