CHALMERS

Know your neighbor: self-organizing data streaming processing in Advanced Metering Infrastructure

Master of Science Thesis in Programme Computer Systems and Networks

THEODOROS KOTSANTIS

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, April 2014

Know your neighbor: self-organizing data streaming processing in Advanced Metering Infrastructure

Theodoros Kotsantis

Examiner: Marina Papatriantafilou

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden April 2014

Abstract

Devices in fields like telecommunication and infrastructure networks as well as telephone and stock markets are sources of continuous data streams. Data streams are usually data created by sensors. These sensors are great in number, for example the sensors that automatically measure electricity consumption in a country can be hundreds of thousands. Also these sensors can vary the rate they create new data. In addition several of the applications that create data streams require close to real time processing of data. For example, having access in real time to the electricity consumption of all the users in one district, we could detect if the part of the network that serves that district is getting overloaded and take on actions that would prevent an outage. So in order to process data streams, we should address three challenges: process a) high volume of data, b) with a fluctuating volume in a c) close to real time fashion.

Stream Process Engines (SPEs) are the systems that emerged from the Database (DB) community and are designed to process data streams in an online fashion. Stream Engines process the data streams as soon as data arrive by applying continues queries. Each data element is processed at most once, without the need to persist the information first. Stream engines can process great volume of data with close to real time delay. Stream engines can run in a distributed fashion but we need to use Load Balancing (LB) protocols, so the stream engines would cope with fluctuations in the rate that data arrive.

One system that could benefit from Stream Processing is the Advanced Metering Infrastructure (AMI), the system responsible for the reporting of end user electricity consumption to utility management office. For this thesis we assume on the AMI there is installed a distributed SPE application. We assume that the AMI that we examine in this thesis uses a hierarchical network to transfer information (i.e. smart meter readings). Distributed SPE systems that exist in bibliography don't provide a LB protocol for a distributed SPE application with hierarchical structure. For this reason we want to develop a LB protocol that can apply to the distributed SPE application of our AMI system.

In this thesis, we developed a load balancing protocol that can avoid overloading and can spread the workload among the entities of distributed SPE application. We studied the characteristic of such a system, developed the load balancing protocol, implemented and evaluate its performance in a simulated environment.

Acknowledgements

With this master thesis project I completed my studies in the Department of Computer Science and Engineering, of Chalmers University of Technology. I want to acknowledge the people who supported me and encouraged me in my studies during these years.

I would like to express my sincere gratitude to my project supervisor Vincenzo Massimiliano Gulisano for all the support, guidance and encouragement he offered me. I would also like to thank my examiner Marina Papatriantafilou for her valuable guidance on this thesis project.

I would like to acknowledge all the professors of the University for the knowledge they transferred to me and the time they devoted for me.

Finally I would like to thank my family for their love and support through my academic life.

Theodoros Kotsantis

Göteborg, March 2014

Contents

1. Introduction

1.1 Background

Over the past decades, the integration of computer chips across the devices that people use on a daily basis, which can range from room temperature thermometers to machinery inside a factory and even sensors in agricultural crop fields, lead to a constant increase in the volume of data that need to be processed and stored. When we add up to the already great amount of data acquired from computer systems and mobile devices, we end up with a tremendous load of information.

Current applications of stock market and electronic trading, cellular network and infrastructure networks for reasons like control, commanding, accounting, fraud detection and others, produce continuous streams of data (data stream [1]). All these applications require a close to real-time processing of the data, in order to be effective. For example, it is important to monitor in real time the electricity consumption of a district in order to detect changes in the consumption that would lead to a blackout and take measures to present it. Hence data processing must be done close to real time.

Combining those two factors together, we end up with a vast amount of data which must be processed within close to real time margins. Traditional computing system solutions, like Database (DB), where we store the information and then process them, are not the most appropriate solution in this context; storing first the information adds unnecessary latency that some systems cannot tolerate. Plus we don't always need to store the information but instead process it and use the outcome [2]. For example, an application that uses the readings from sensors to determine the current condition of the electricity network by detecting values above a threshold does not need to persist all the data. To overcome the limitations of DBs for this kind of applications, Stream Processing came in use.

Stream Processing is a way that we can process data streams. In Stream Processing we have two main components. The first component is the devices or applications that produce the data streams and the second component is the application responsible to process the data streams. For the first component we have the devices that continuously produce new data. For example, the devices that periodically measure the electricity consumption in each house. The data from each device are organized in data streams, with each data stream a sequence of tuples that have the same schema. For the second component we have the application responsible to process the data streams. Here we have a Directed Acyclic Graph (DAG) of query operators that form the "continuous query" [3] and use them to process the data streams. The operator applies a function on each tuple individually (stateless operator) or to group of tuples (stateful

operator). The tuples from each data stream are fed to the operators that are responsible to process them (i.e. the operators which form a continuous query). In this way we have each tuple flows through the query operators and is processed on the fly.

Stream Process Engines (SPEs) are examples of application that can be used to run stream processing applications. In a SPE the query operators from the continuous queries stay loaded on the main memory. As soon as a new tuple from a data stream arrives, is loaded directly on the main memory and it gets processed. Because everything is done on the main memory, SPEs are able to process data streams with minimal delay in comparison to DB systems.

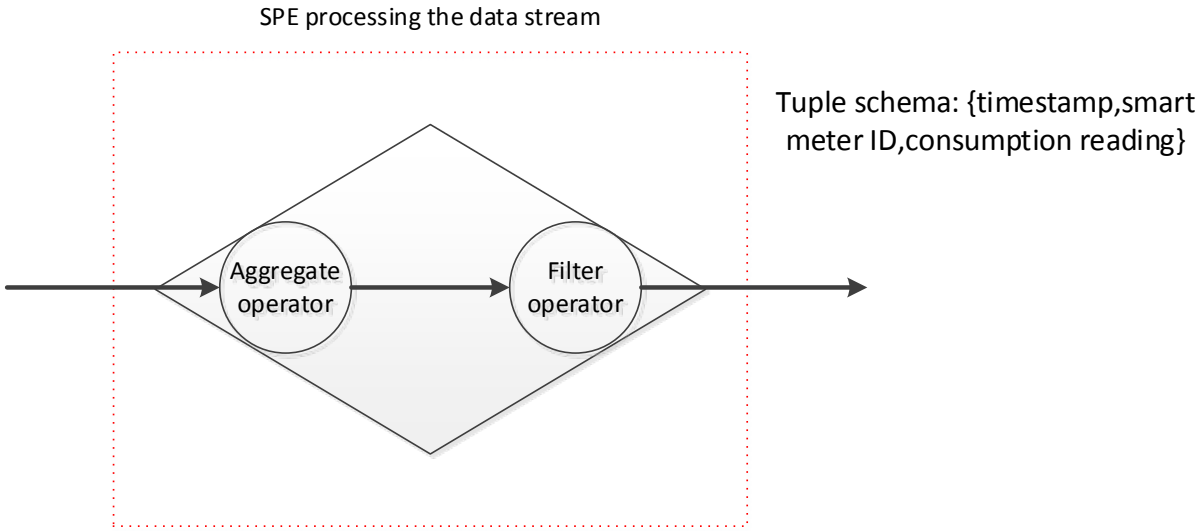

Figure 1 – A continuous query processing a data stream on a SPE

Figure 1 is an example of a one continuous query loaded on a SPE application and presents graphically how stream processing works. The arrows represent data streams and circles represent query operators, with the operators loaded on the main memory of the SPE. The data streams are related to energy consumption readings of the users in a district. The tuples have three attributes: timestamp, smart meter ID and energy consumption reading. Here we have a DAG of two operators that form a continuous query which spots customers whose aggregated energy consumption over a day exceeds a given threshold. The tuples for each user are aggregated in the first operator, using a window of one day. Then the output tuples from the first operator are fed to the second operator. The second operator filters out the tuples that have value below a given threshold and forwards the rest.

SPEs began as centralized systems (Aurora [4], STREAM project [5], TelegraphCQ [6]) and evolved to distributed solutions [7] [8]. The main difference between centralized and distributed SPEs is that in the first all the processing is done in one node (i.e. all the operators that belong to the same query are all deployed on the same node), while on the second one the operators that belong to the same query can be deployed to multiple nodes (inter-operator parallelism)

[7]. On figure 2 we present an example that shows this difference between centralized and distributed SPEs.

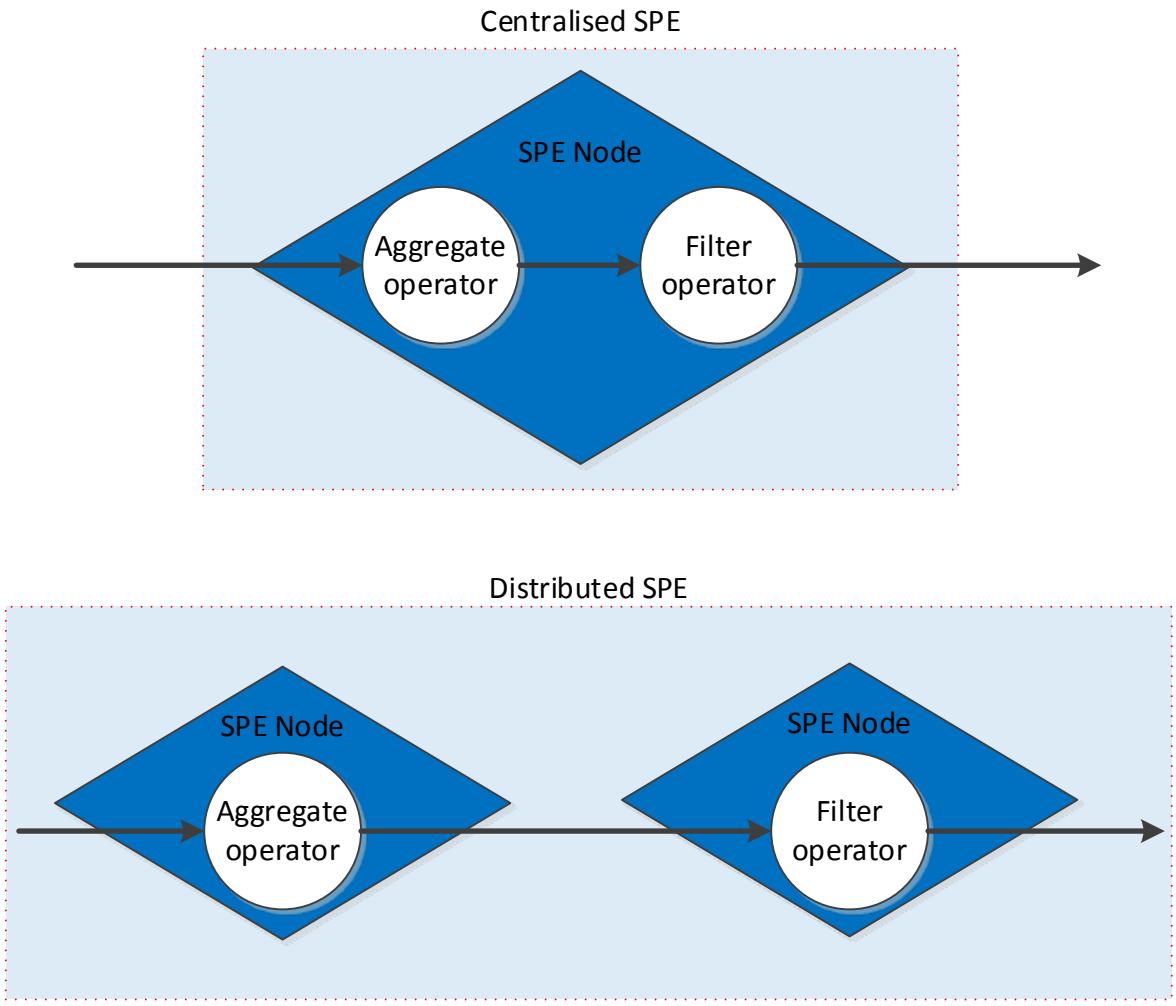

Figure 2 – The same continuous query deployed on a centralized SPE and a distributed SPE

On figure 2 we can see how the same query from figure 1 could be deployed to a centralized SPE system and how could be deployed on a distributed SPE. On the centralized SPE all the operators that belong to the same query are deployed on the same node. The processing is done in the same way that we described before (description for figure 1).On the distributed SPE the first operator from the query is deployed on the first SPE node and the second operator from the query is deployed on the second node. Here the data from the readings are routed to the first node where are processed by the aggregate operator and the output tuples are routed to the second node where they are processed by the second operator.

The main disadvantage of centralized SPEs solutions is that the whole system is bounded to the processing capabilities on one SPE node. Distributed SPEs systems have the advantage that they can scale up better when we need to add more resources to the system in order to cope with the increase of the load that must be processed.

An application of distributed SPE can be used with Advanced metering infrastructure (AMI) systems. AMI is a system part of the Smart Grid [9] and is a system responsible for measuring periodically the end users electricity consumption and report the information remotely to the utility management office. An AMI system includes the smart meters, the AMI communication network and the utility office. First, the smart meters are the electronic devices that are responsible to measure the electricity consumption at the user end point. Smart meters have bidirectional communication capacities and can report periodically the meter's readings. Second, the AMI communication network is the infrastructure responsible to transfer the information, for example the readings from the smart meters to the management office. Third, the utility office is where the readings from the smart meters are collected and get further processed, for example to calculate billings among others.

For the purposes of this thesis we are examining the scenario where the AMI communication network is a hierarchical network. The network consists of devices called Meter Collector Units (MCUs) which are responsible to forward the energy consumption readings to the utility office. Each MCU is responsible to handle data at different levels; for example an MCU is responsible to collect the data from the smart meters at a neighborhood and forward them on the MCU that is above it in the hierarchy (i.e. a MCU on higher level which is responsible to handle the data from the MCUs at multiple neighborhoods and so on). Last we have to specify that a MCU cannot reroute the data to other MCUs, but can only forward data to the MCU that is responsible to collect them.

Using the smart meters and the infrastructure at the AMI communication network, we can run a distributed SPE in order to address various issues related to AMI (accounting, monitoring and management). Here we examine a system which consists of: First, the smart meters that are the devices that create the data streams. And second the AMI communication network, on which is installed a distributed SPE system. On each MCU we assume that there is installed an SPE application capable of processing the data from the data streams that smart meters create. All MCU can cooperate to form the distributed SPE system. All the processed data are eventually forwarded to the utility office.

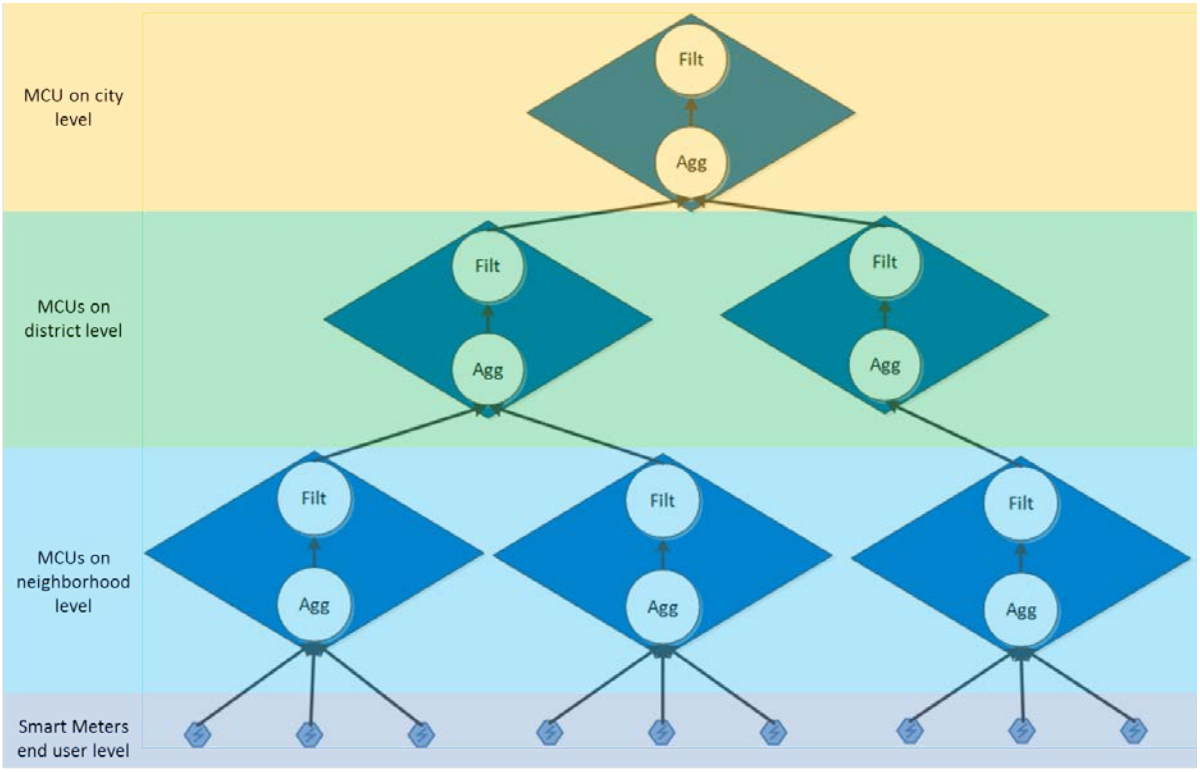

Figure 3 – MCUs monitoring the average consumption and filter out suspicious values at different levels: end user, neighborhood and district

On figure 3 we present an example of the system that we are going to examine for the purposes of this thesis. This example is about how we could monitor the average consumption and filter out suspicious values at different levels: end user, neighborhood and district. The arrows represent data streams, each diamond represents a MCU on which we can execute a SPE application and the circles represent query operators. At the bottom of the figure we can see the entities that represent the smart meters at the end user lever. One level above, the neighborhood level, we group the smart meters in neighborhoods and for every neighborhood we have a MCU that collect and process the data streams that originate from the smart meters. Above the neighborhood level is the district level, where each MCU collects and process data from the data streams that MCUs on neighborhood level create. On top of the picture we have a MCU responsible to collect and process data from the MCUs at district level. On each MCU we can deploy operators, with a DAG of operators to form a query. Here each query consists of 2 types of operators an aggregate and a filter operator. The aggregate operators can calculate the average consumption on different level and the filter operators can filter out the values that are below a threshold. For example, at a MCU on neighborhood level, we can aggregate the readings from the smart meters and filter the suspicious values for each end user.

The problem we try to address with the applications of SPEs (in our case the distributed SPE at the AMI communication network) is that the rate in which the data is produced can vary by

time. For example, an electricity company can apply different accounting policy on specific time of the day, which requires the smart meters to measure the consumption more often. This means that instantly the MCUs have to process greater volume of data without delaying the delivery of results.

This high volume of data with varying rates creates two challenges for us: first there can be MCUs which might not be able to keep up with the load of information that they have to process, while other MCUs might not perform to the fullest. The second one is related to the fact that you have restriction about where each operator of the query can be actually deployed. For instance as we can see on the example of figure 3, we could not deploy the aggregate in charge of aggregating the data of an entire district (aggregate from the MCU at city level) at an MCU that is not "high enough" in the hierarchical network (for example a MCU at neighborhood level) to have access to all the aggregated reading of an entire district.

As we mentioned previously distributed SPEs provide the means to deploy operators at arbitrary SPE nodes (example on figure 2). In addition they can also provide the means to transfer operators between arbitrary SPE nodes, i.e. change the node on which an operator in deployed on run time. On figure 4 we can see an example of the operator transfer. Here we can see a distributed SPE system with two SPE nodes: 1) at the beginning all the operators from a query are deployed on one SPE node and the results are forwarded to an application. On the same time, at the second node there are no deployed operators. On an operator transfer process, 2) the node that possesses both the operators would stop temporarily to forward the tuples to the filter operator. 3) The filter operator would get deployed on the second node. 4) The first node would resume forwarding the tuple from the aggregate operator, but this time it would forward then to second node which is now runs the filter operator. At this point the second node would forward the results to the application. In this thesis we suppose the actual operator transfer procedure is provided by the distributed SPE.

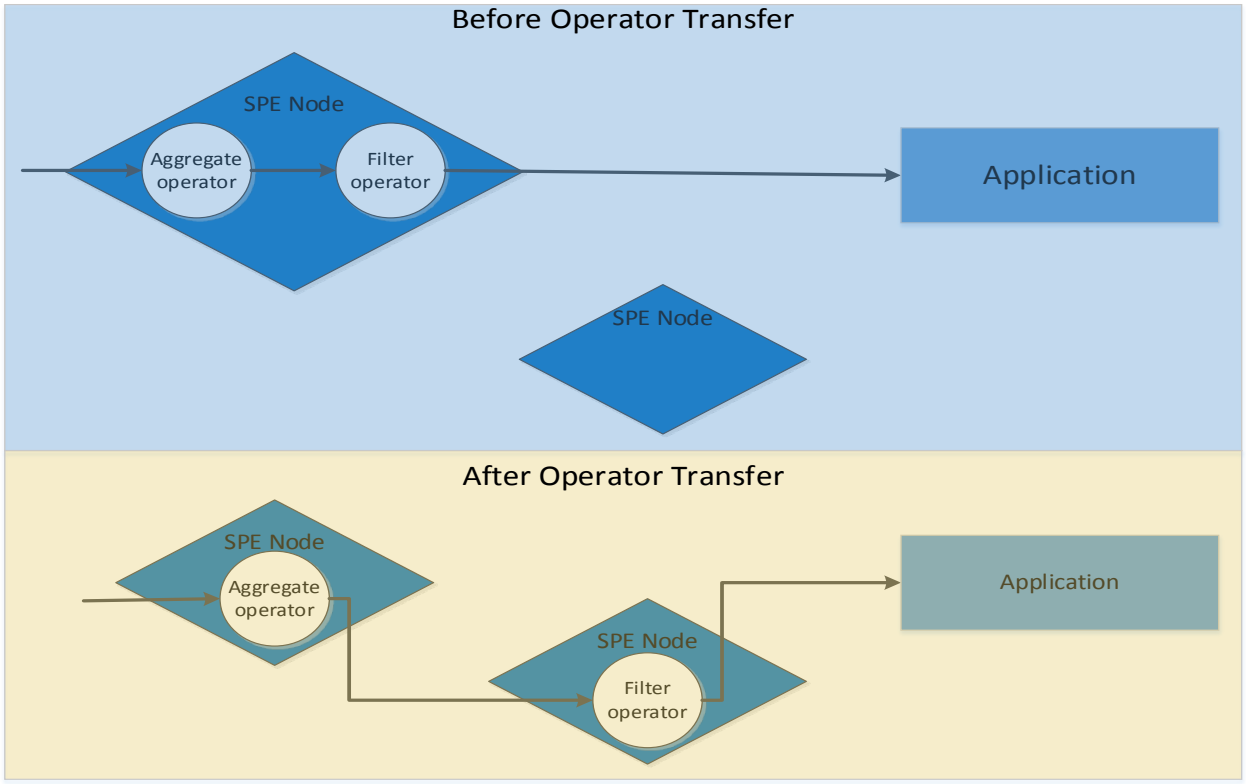

Figure 4 – Operator transfer example

For the purpose of this thesis we want to use the functionality to transfer operators between the nodes of a distributed SPE system in the system we examine, the distributed SPE application on AMI. We want to use operator transfer in order to provide load balancing capabilities in our system. Our goal is to cope with the fluctuation in the volume of the data streams created by the smart meters.

1.2 Short problem statement

In the AMI system there can be time periods which the volume of data increase to such a degree, that the MCU nodes don't have the resources to run all the operators deployed on them with the excessive load. Some nodes can get overloaded (a node can have so high memory and CPU utilization that it cannot process the data in the rate they arrive) and lose the time constrains that they have to produce the output, while other nodes are underloaded because they receive no data to process.

The aim off this thesis work is to develop a load balancing protocol to avoid overloading of nodes and provide load sharing between the MCU nodes. The detailed statement of the problem as well as the evaluation criteria, are presented on section 2.

1.3 Goal

This report deals with the creation and evaluation of a load balancing protocol that could potentially run on the MCU nodes of the AMI communication network. We suppose that we have a fixed group of MCU nodes which can run SPE applications. When the performance of node gets outside some predefined margins, the node will autonomously share the workload with the adjacent peers. We plan to achieve that by designing and developing a Load Balancing protocol, which will be used by the adjacent nodes to negotiate about which actions they are going to perform.

To our knowledge there is no other load balancing protocol developed for a distributed SPE application with the SPE nodes hierarchically organized. The nodes in the distributed SPE that we examine should autonomously adapt to the variations in the tuple creation-rate with the challenges that: first, the data streams cannot be rerouted and will always be routed through the same nodes. And second, we have restrictions about where each operator of the query can be actually deployed. Operators can be deployed only to nodes that have access to all the data streams that the operator uses.

2. System description and problem modeling

In this chapter we begin by making an introduction to data streaming processing and AMI systems, the terms and systems that we are going to use for this thesis. We continue with the second section where we present the overview of the distributed SPE system that we are going to examine. On the third section, we present a detailed description of the problem that we want to address with this thesis. On the fourth section we finish this chapter with the method that we are going to evaluate our proposed solution.

2.1 Terms definitions

2.1.1 Data stream

Data stream "is a sequence of data items that we can access only once" [1]. Data streams are usually related with applications that produce high volumes of new data in real time; telephone records, earthquake sensors, electricity smart meters, stock exchange records are some typical examples. A data stream consists of tuples with the same schema. For example, on an AMI system the tuples can have three attributes: timestamp, smart meter ID and energy consumption reading.

Most applications creating data streams have the common characteristic that there are time periods where the rate that they create new data fluctuates. The tuples are usually created periodically by devices or sensors (for example the smart meter on a house reports the readings every two hours, all year long) but this is not always the case. For instance the same smart meter from the previous example, in the event of an outage could start to report its condition every half an hour, because the extra data could help to narrow down the failure that caused the outage on the first place.

2.1.2 Queries and Operators

Data streams are essentially sequences of tuples having a timestamp to specify the time point that they were created. Like the tuples organized in tables in a DB system, the data streams can be processed by applying queries on them. But unlike DB systems where a query is only issued when the user decides to retrieve information from an instance of the DB, the SPE loads the query on the main memory and the query consumes each new tuple that arrives.

Like in traditional DB systems, queries consist of operators. In stream processing each query is usually presented as a DAG of operators. Most common operators can be divided in two categories, the stateless operators (filter, map, union) and the stateful operators (join, aggregate, sort) [4]. Stateless operators perform computations separately to each input tuple, without taking into consideration the values of the previous or sequent tuples. Stateful

operators on the other hand, perform computations on window of input tuples.

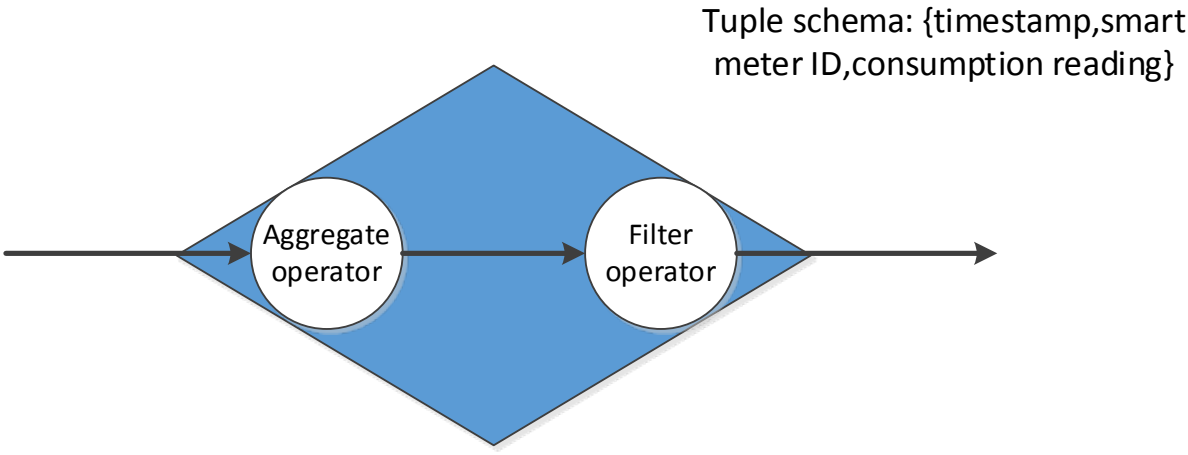

Figure 5 - Query which spots customers whose aggregated energy consumption exceeds a given threshold

In figure 5 we can see an example of a query which spots customers whose aggregated energy consumption over the last 6 hours exceeds a given threshold loaded. The arrows represent data streams and circles represent query operators. The data streams are related to energy consumption readings of the users in a district. The tuples have three attributes: timestamp, smart meter ID and energy consumption reading. Here we use two operators that form the continuous query, an aggregate and a filter operator. The tuples for each user are aggregated in the first operator, using a window of 6 hours. Then the output tuples from the first operator are fed to the second operator. The second operator filters out the tuples that have value below a given threshold and forwards the rest.

2.1.3 Stream Process Engines

Stream Process Engines (SPEs) are examples of application that can be used to run stream processing applications. It is the SPEs where we deploy on and run the query operators from the continuous queries, to process the data streams. In the following two subsections we present two versions of SPE systems, the centralized SPEs and then the distributed SPEs.

2.1.3.1 Centralized Stream Process Engines

Centralized Stream Process Engines [4] [5] [6], like all SPEs, are computational systems that can process data stream without storing the data first. Storing data before we process them adds time latency that certain applications cannot tolerate. Furthermore, in most cases it is more essential to store the output (processed) data instead of the input data streams. As soon as the tuples arrive they are loaded directly on the main memory where they stay until they are processed by the according operator(s). This method provides a higher processing capacity compared to a traditional DBs, which first retrieve the stored data, load them to main memory and then process them.

On centralized SPEs there is only one processing entity and all the operators are deployed in the only SPE node. The data streams, which begins from the source (for example an sensor), are routed to the SPE node where is used as an input on the first operator of the query and the output from the first operator is used as an input on the second operator and so on.

The problem with the centralized SPE solution is that not all operators that belong to the same query have the same processing latency. An operator can become a bottleneck if the rate that new tuples arrive increase. For example we have two operators on the same query deployed in the SPE node. The first operator process tuples in 1 msec and the second operator process tuples in 3 msec. If new tuples arrive every 3 msec the node can execute query with no problem. But in the case where the new tuples arrive every 1 msec, the second operator becomes a bottleneck.

When an operator becomes a bottleneck, for example during the periods that new tuples arrive with increased rate, there are two main solutions: either we can apply load shedding strategies (i.e. drop selected tuples from the data streams [4]), or we can replace operators from the query which becomes bottleneck with other operators [7]. In both cases we would need to drop tuples from the data stream, which would lead to changes in the accuracy of the output result. Load shedding affects the result because we drop tuples from the input data stream in order to decrease the workload. Replacing an operator can be performed online (replace the operator without stopping the processing in the rest operators). If the operator that creates the bottleneck is stateful, when we replace it we will also affect the output result because the older tuples (that have been already processed) are not used for the results with the new operator.

To summarize, centralized SPEs are applications that we can process data streams, process high volume of data in close to real time margins, but they have problem to adapt to rate fluctuations.

2.1.3.2 Distributed Stream Process Engines

SPE can be centralized or distributed. In central SPEs the process of all the data streams is done in a central system with increased processing capabilities. But a centralized solution of SPE has the disadvantage that it does not scale up efficiently when the volume of data stream increase. Distributed SPEs [7] [8] are the evolution of SPEs.

Distributed SPEs address scalability issues of centralized SPEs by distributing the operators that belong to the same query to different SPE nodes. DSPEs can deploy multiple SPE enabled computer clusters where we have the option to move operators inside the same cluster as well as transfer operators of queries between different clusters. With this option distributed SPEs have the ability to scale effectively, be applied on heterogeneous clusters, and share the workload among the autonomous systems.

In most cases the processing entities (SPE nodes) of the distributed SPEs are running more than one operator simultaneously. When a node becomes overloaded (has very high utilization of CPU, memory, network usage or high processing time) there is an option to move one or more operators, so these operators would run on another SPE node. The SPE nodes of a distributed SPE system have the ability to transfer operators among them.

When we refer to operator transfer, we mean to deploy an operator, which is already deployed at one SPE node, to a new node and redirecting the data stream to the new node. On figure 6 we can see an example of this procedure. The procedure is performed between two autonomous SPE nodes. In this example both of the SPE nodes have access to the same data stream. At the initial phase, Node 1 has deployed on the main memory operator A and uses it to process the data stream. On the same time node 2 only forwards tuples from the data stream. At the middle phase, node 2 stops forwarding tuples to node 1 and begin to buffer the new tuples. On the same time node 2 deploys operator A on its main memory. Node 1 processes all the tuples which has already received and forwards all the processed tuples. At the last phase, node 2 begins to process data stream and node 1 just forwards the processed data stream. In the case where the operator is stateful, when the node 2 resumes processing the results from the operator would not be completely accurate until the window of the operator is full.

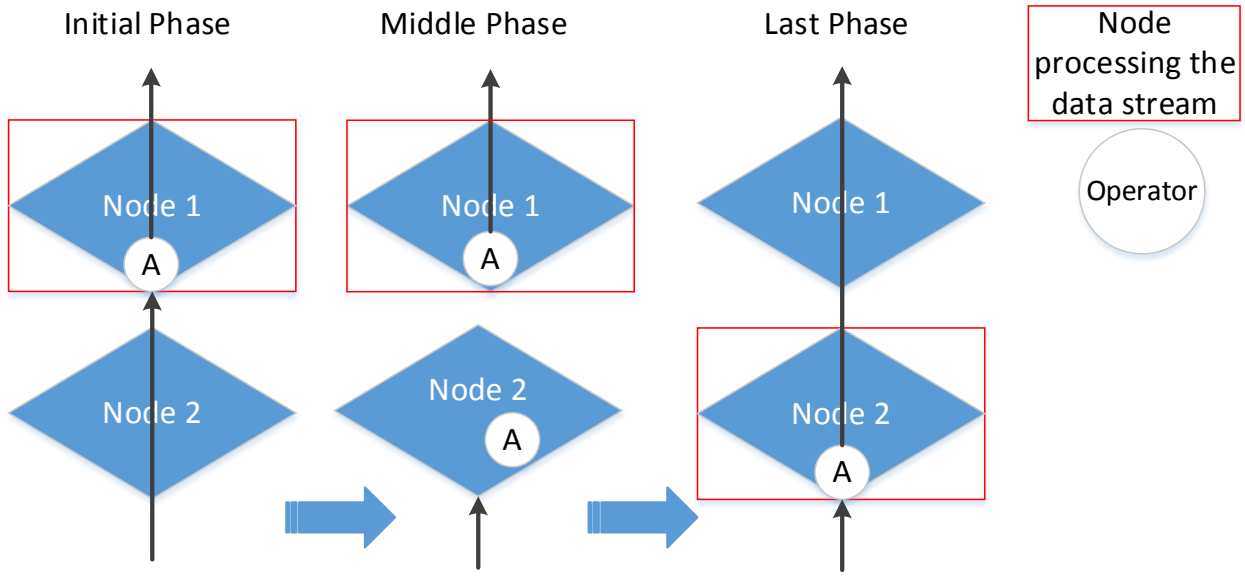

Figure 6 – Single operator transfer between adjacent nodes

In general, distributed SPEs can adapt with the changes in workload created by fluctuations in the rate of data streams on runtime. Transferring operators in-between the SPE nodes of a distributed SPE system and redirecting the data streams, distributed SPEs can move part of their workload to other SPE nodes when they get overloaded.

2.1.4 Advanced Metering Infrastructure

The term Smart Grid refers to the integration of applications related to communication technologies and data processing on top the electricity power grid. Smart Grid is an overlay network which connects the power stations, transmission network, distribution network and smart meters at the consumers' end point [9].

Advanced Metering Infrastructure (AMI) system is responsible for measuring the consumers' electricity consumption and reporting that information periodically to the utility management office. In general AMI systems consist of 3 major entities: smart meters, AMI communication network and utility office. The smart meters measure the end users consumption and using the AMI communication network, the information is delivered to utility office.

Smart meters are the devices that calculate the power consumption at the consumers' site. They have telecommunication capabilities for providing close to real time billing services and reporting back to the electricity provider. Smart meters use bidirectional communications channels where in addition to sending the billing reports, they can receive commands. For example, while a smart meter sends reports every 2 hours, it can receive a command to send reports every 15 minutes instead.

The AMI communication network consists of units with telecommunication capabilities, and is responsible to transfer information messages (for example, reports and commands) from and toward the smart meters. The AMI network is combination of Home Area Networks, Neighborhood Area Networks and Wide Area Networks [10].

Electricity companies plan to apply response demand and dynamic pricing policies. The end user would pay different price rates depending on when he uses electricity. For example, the electricity rate would be higher when the electricity network is overloaded. This requires to change the rate that smart meters report; the smart meters should report more frequently at specific time periods. AMI systems allow electric utility companies to control the rate that the consumption reporting is done. But by changing the rate that smart meters report, we change the volume of data that system must process. In addition these services require the AMI network to provide metering in close to real time [11].

The integration of distributed SPE applications on the AMI system is a solution to achieve this goal: distributed SPEs could potentially process big amount of data on close to real time margins and can adapt to variations in the creation rates. For example, in hours with high electricity demand and constantly changing charging rates, the smart meter can receive a command to increase the frequency in which the consumption reports are sent. This automatically increases the processing workload that must be processed with the existing processing infrastructure. If we could install distributed SPE nodes on the AMI communication network, we could provide a solution.

2.2 Problem modeling

2.2.1 System Overview

For the purposes of this thesis we assume that on the AMI communication network there are nodes, which have the ability to run an application that allows for distributed execution of data streaming queries, in order to process the data streams from the smart meters. More specifically the AMI network consist of MCUs which are hierarchically organized (figure 3): On the far bottom are the smart meters which are constantly producing new data tuples (data stream source). Above the smart meters is the AMI communication network. We assume that MCU nodes are able to run distributed SPE applications (for simplicity from this point on, we are going to refer to the SPE application running on each MCU as SPE node). The SPE nodes are responsible for routing and processing the data. Each SPE node has a group of operators which perform different operations on the data; i.e. operators process the tuples who compose the data stream.

As we already mentioned, we assume that the AMI network is hierarchical (figure 3). Each MCU is responsible to handle data at different levels; for example an MCU is responsible to collect the data from the smart meters at a neighborhood and forward them on the MCU that is above it in the hierarchy (i.e. a MCU on higher level which is responsible to handle the data from the MCUs at multiple neighborhoods and so on). We have to specify that a MCU don't reroute data streams to other MCUs, but can only forward data to the MCU that is responsible to collect them. MCU theoretically could re-route streams, but we don't consider this option because it would lead to a considerable increase of the latency (MCUs are usually embedded devices with limited resources).

In our system the SPE nodes are able to transfer the processing of data from one node to another, for example when an SPE node gets overloaded. As we discussed before (section 2.1.3.2) the SPE nodes are able to transfer operators of queries they possess to other SPE nodes. On the same time they forward the data stream to the node that would do the processing (figure 6).

At this point we have to specify that we assume that we take into account fluctuations on the creation rate that are still "manageable". That is, we can have a load increase that is still processable given the resources we have (after balancing the load among them). If the overall load grows beyond the available resources we would need other ways of dealing with it (i.e., elasticity) which is not in the scope of this work.

When a SPE node decides to transfer an operator to another SPE node, it can transfer an operator only to direct adjacent neighbor nodes. More specifically, a SPE node can transfer an operator either to a node from which receive data stream (for simplicity we will call this SPE node, child node), or to the node that forwards a data stream (parent node).

Due to the hierarchical AMI network not all SPE nodes have access to all data streams. Therefore only specific transfers of operator(s) are feasible with each configuration of the system. For example on figure 7 we can see that we can't transfer operator A, to one of the child SPE nodes. On the next section we are going to present the feasible operator transfers, taking into consideration the limitations of our system.

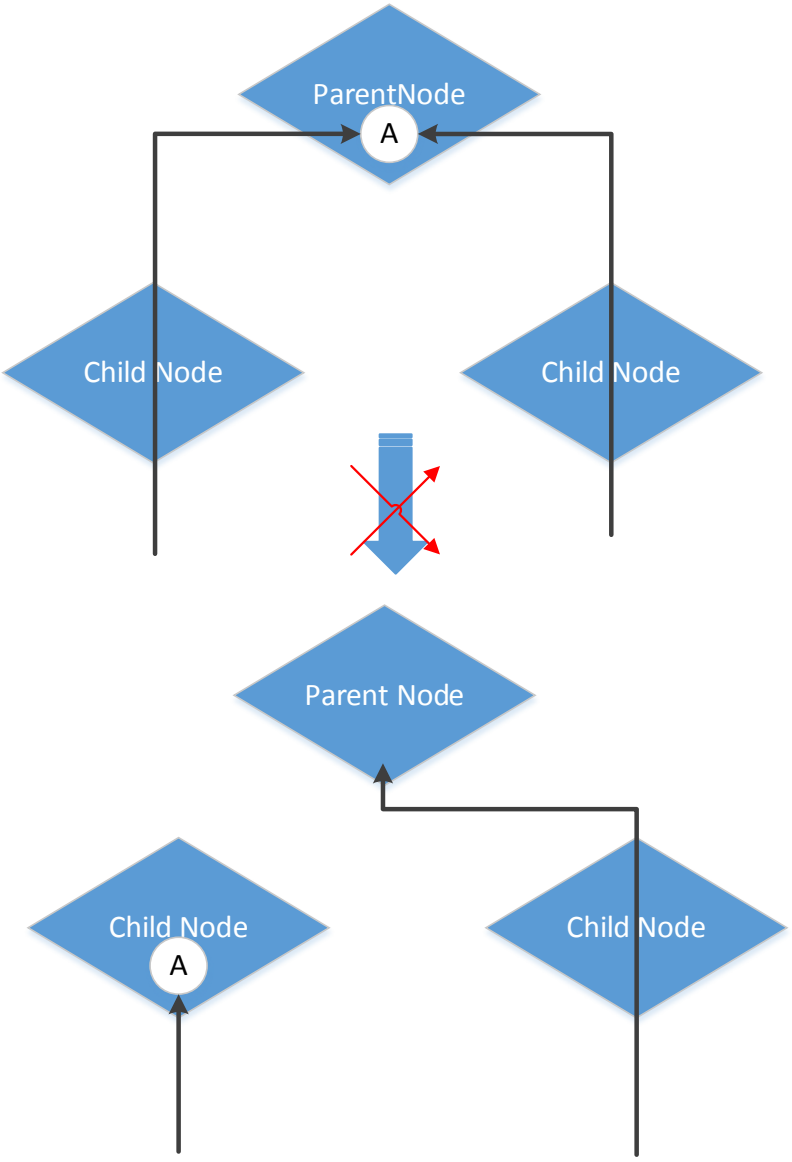

Figure 7- Unfeasible Operator Transfer

2.2.2 Possible scenarios of operator transfer.

When a SPE node is processing a data stream, it is using as input the data that it gets from its child(ren) node(s), it is processing them and then it is feeding the output to its parent node. Each operator running in a node has two sets of relationships that affect it:

- Which is(are) the operator(s) that is(are) producing the input stream(s) and which is the operator that process the output stream.
- Which are the nodes from which the input stream originates and which is the node that the output stream ends; for example, the operator which produces the input stream is running at a child node, and the operator processing the output stream is running on the same node.

The relation between an operator running in a node and input/output data streams (i.e. originating and destination nodes) arise limitations in the feasible operator transfers; only specific operators can be moved between specific adjacent nodes. In our system we observed three main scenarios that describe the relations of the operators running in a node and the feasible operator transfers.

First scenario, a node has one parent and one child node and the operator running on that node has an input stream originating from the child node. In this scenario there are two operator transfer options:

- The operator can be transferred to child node.
- The operator can be transferred to parent node.

Second scenario, where the operator running in the node is dependent to another operator on the same node. For example operator A can create output that is used by operator B running on the same node (figures 8 and 9). In this case we have four operator transfer options:

- Move both operators to child node.
- Move both operators to parent node.
- Move one operator to child node.
- Move one operator to parent node.

If we decide to transfer only one operator to another node, we must first check if this is feasible. We either can move operator A to child node or move operator B to parent node (figure 8 and 9 accordingly).

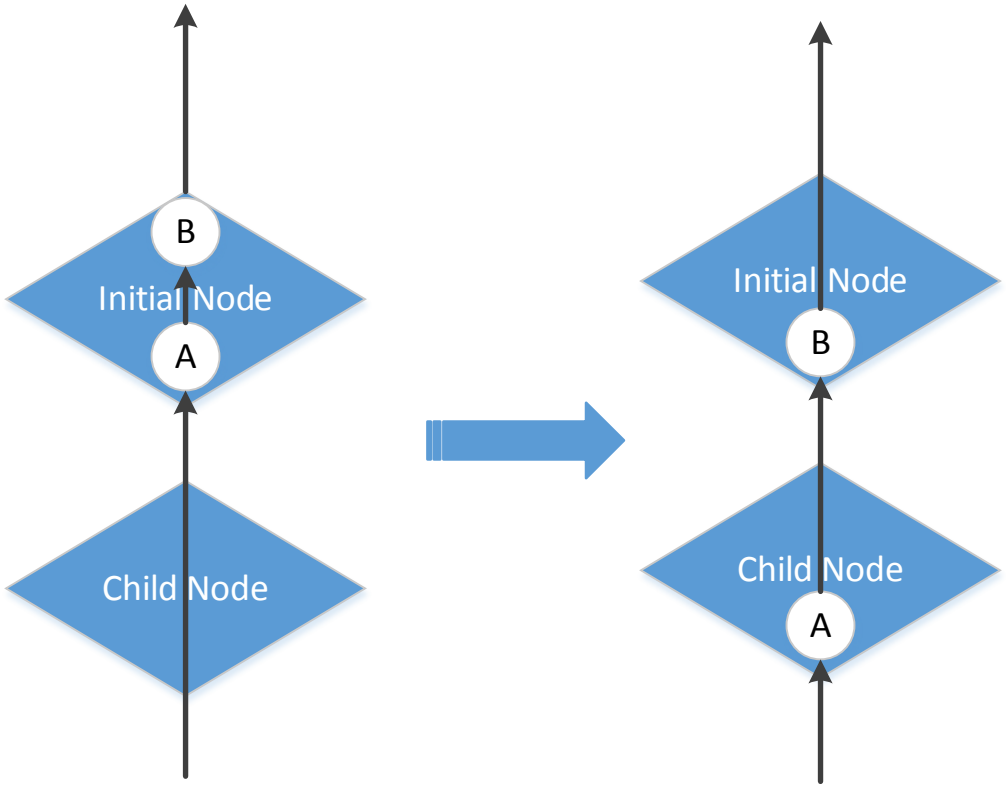

Figure 8- Operator transfer to child node when there are two depended operators on the same node

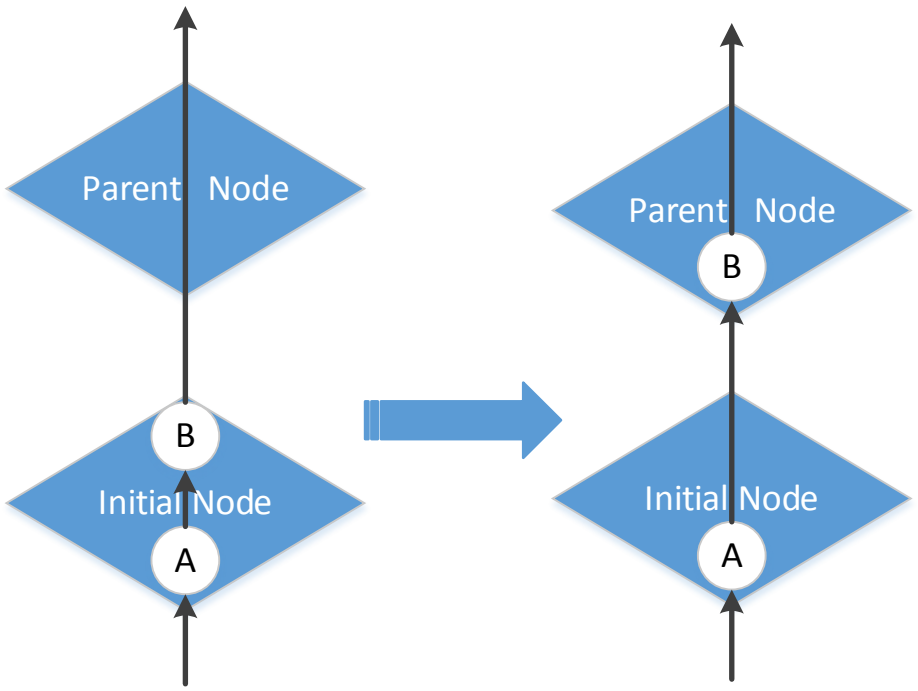

Figure 9- Operator transfer to parent node when there are two depended operators on the same node

Third and last scenario, an operator is feed by multiple children nodes. There can be operators that use as input data streams which origin from different children nodes. For example in figure 10, the initial node is gathering three different data streams and processes them applying the same operator on them. In this scenario, if we want to transfer the operator from the initial node we can only move the operator to the parent node.

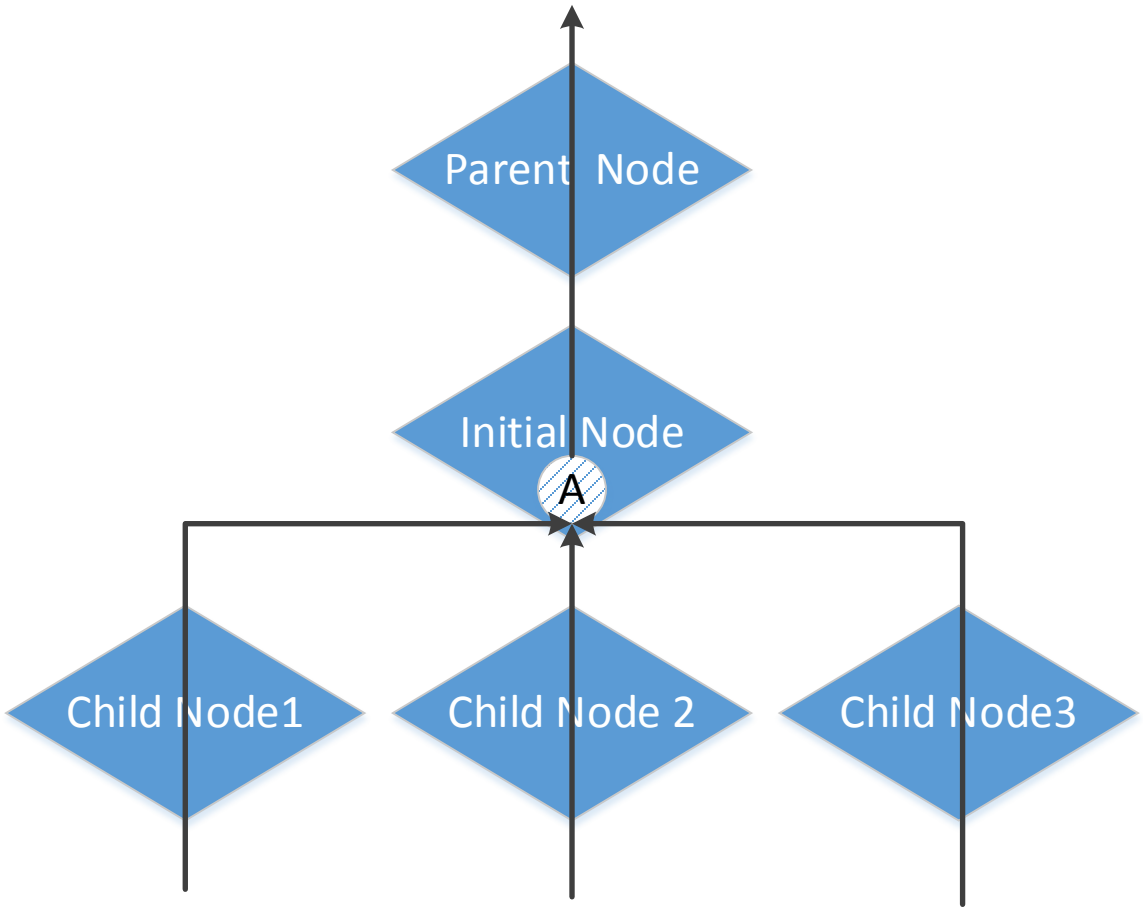

Figure 10– Multiple input streams from different nodes

2.3 Detailed problem description

On the AMI systems there are fluctuations at the rate that new tuples are created by the smart meters. In order to process the data stream from the smart meters without reducing the accuracy of the output result, the application that is responsible to process the data should adapt on runtime and process the extra workload.

For this thesis we assume that we use a distributed SPE application, installed on the AMI communication network, to process the data streams. We assume that the distributed SPE application is installed on a fixed number of MCUs, MCUs that form the AMI network, since we assume that we cannot add more MCUs in the network. All MCUs are hierarchically organized with each one of them to have access and be responsible to process data streams, on different level.

On each MCU we assume that is running a SPE application (SPE node) and on the SPE node we can deploy operators to process the data streams. In general we consider that a SPE node has deployed a number of operators and is processing the tuples with no delay. Due to the fluctuations at the data streams, there can be time periods that the number of tuples increase to such a degree that the SPE node does not have the recourses (for example the main memory) to process all the tuples on time. For example in a case that there are deployed three operators on the SPE node, the node has the resources to process the tuples that belong to two of the operators on time and begin to delay the results for the third operator. The solution to this problem would be to transfer one (or two) of the operators to another SPE node that has the resources to process it with no delays.

Distributed SPE applications provide the option to transfer operators to another SPE node but the process is not straightforward. In the AMI network that we examine, the MCUs are hieratically organized and they are not allowed to reroute data streams. These conditions create limitations, on where we can transfer an operator; the SPE node which would receive and deploy the operator, should have the adequate resources and access to the data stream(s) that are required, in order to run the operator.

The operator transfer capability allows us to spread the workload among the SPE nodes. With this master thesis we want to develop a Load Balancing protocol, which would avoid overloading of SPE nodes and that would also provide load sharing between the SPE nodes, in order to counter the data load fluctuations at the AMI system. We want to use the LB protocol in order to decide which operators would be transferred, between the SPE nodes installed on the MCU of the AMI communication network.

2.4 Evaluation method

For the purposes of this thesis we use memory utilization of each SPE node to measure the performance of each node. We assume that each operator deployed on a SPE node occupies a percentage of node's main memory. When the rate that an operator receives tuples increase, we consider that the size the operator occupies on the main memory increase. When the rate that tuples arrive decreases, the size the operator occupies on the main memory decreases. So in order to determine the performance of each SPE node, we measure its memory utilization and if it's above a predefined threshold the node is overloaded.

In order to check the performance of our protocol, we are going to create simulated environment; we assume that we have an AMI communication network with MCUs and on each MCU we are able to run SPE node. We consider that the distributed SPE system that the SPE node create, provide as the option to transfer operators between the SPE nodes.

In our simulations, we have a fixed number of SPE nodes, on which we assume that we have deployed operators. We would artificially create fluctuations in the rate that tuples arrive and would have two simulations run: one without using of LB protocol and one using the protocol. We plan to evaluate our results comparing the number of overloaded node for each pair of simulation run. We are going to evaluate our protocol according to if the protocol reduces the number of overloaded nodes and if it reduces the time that nodes stay overloaded.

3. Design

3.1 Protocol

In order to avoid overloading and perform load balancing among the SPE nodes of our system, we developed a Load Balancing protocol which regulates the operator transfers between the SPE nodes. The outer goal with this protocol is to distribute the excessive processing load, generated by the fluctuations in tuples creations in the smart meters.

For the purposes of this thesis we measure the performance of each node according to its main memory utilization. The idea is that each node has as a target to perform normally, i.e. within two predefined margins of memory usage. When a node performs outside these margins, due to fluctuations in the volume of data it is responsible for processing, the node transfers operators towards its adjacent nodes in order to regulate its memory usage.

In this section we present the LB protocol that we designed. We begin with a general introduction of the three main actions that are performed by the protocol and we proceed with a more detailed description on the following sections.

As we presented on section 2.2, due to the characteristics of the AMI network that we examine, there are limitations on where an operator can be deployed; an SPE node must have access to all the data streams that are used as input for an operator, in order to run the operator. Before a SPE node begin the procedure to transfer an operator to one of its adjacent nodes, the SPE node must to know in advance, which are the adjacent nodes that are allowed to run the operator. In order to achieve it, we created an algorithm (operator selection algorithm) that examine all the operators deployed in one SPE node, and for each operator returns the adjacent SPE nodes that each operator can be transferred.

Our protocol performs three main actions: a) the protocol uses the operator selection algorithm to determine which operators can be transferred to which SPE node. The output of this algorithm is a list of elements called operator transfers (section 2.2.2). Each operator transfer element consists of the name of the operator and the ID of SPE node the operator can be transferred. For example, if at one node there is deployed only one operator that can be transferred either to parent node, or to child node, the operator selection algorithm would produce two elements: both of them would contain the name of the operator, but would differ on the ID of the SPE node that are referring(more details on section 3.2). The operator selection algorithm only examines if one adjacent SPE node has access to the data streams that are needed to run an operator and we don't examine if the SPE node has the resources (free main memory) to deploy the operator.

Actions two and three are connected with the memory utilization of each SPE node. b) When a SPE node has memory utilization above a predefined threshold, the node is consider overloaded and begins action two: the SPE node begins a negotiation round with its direct adjacent node, in order to transfer operator(s) that has on its main memory to adjacent nodes. c) When a SPE node has memory utilization below a predefined threshold, the node is consider underloaded and begins action three instead: the SPE node begins a negotiation round, in order to take on operators from its adjacent nodes.

The LB protocol is distributed and separately each SPE node uses its' local memory utilization as the only parameter to take decision for the protocol. All nodes can exchange information messages and transfer operators only with their direct adjacent nodes.

Each SPE node runs locally a copy of the same algorithm (operator selection algorithm, action one). This algorithm is taking as input the operators deployed on the node and creates a list with all the possible combinations of operator transfers (more details on section 3.2). With a period T, the node is checking the level of its memory usage. If the node's memory usage is outside the predefined margins, the node begins negotiations with its direct adjacent nodes using the LB protocol.

When the node needs to decrease its memory usage (action two), the node sends negotiation messages to its adjacent nodes, which contain the operator transfers created by operator selection algorithm. The nodes negotiate using the operator transfers and if they finally reach to an agreement, an adjacent node would deploy the operator indicated at the operator transfers. After this point the adjacent node begin to process the data streams connected to the operators.

When the node needs to increase its memory usage (action three), the node sends negotiation messages to its adjacent nodes, which contain the memory size of operators that the node can take on. The adjacent nodes reply with their operator transfers that they can provide. Then the node selects the operator transfers that it can take on and deploy the according operators. After this point the node begins to process the data streams connected to the operators.

3.2 Operator selection algorithm

In order to decide which operators can be transferred from one node to another, we developed the operator selection algorithm. In this algorithm we calculate all the possible combinations of operator transfers for this node (from this point and on, we will refer to them as Available Transfers) that we use in each run of the LB protocol. The algorithm decides which operators can be transferred to a parent node and which to a child node. It also decides if two or more operators must be transferred together (with other operator) and to which node.

The algorithm examines each operator separately and checks which, is its downstream operator and which, is its upstream operator. The algorithm also examine if those operators are deployed in the same or different nodes. For example, in the case where a node holds two operators, A and B, while operator A depends to operator B (depended operators on the same node figures 8, 9). The algorithm will produce four combinations of Available Transfers: a) transfer operator A to child the node, b) transfer operator B to the parent node, c) transfer both operators A and B to the child node and d) transfer both operators to the parent node. Algorithm is repeated only when we add or remove an operator from the SPE node.

Table 1 – Classes Operator, Transfer and Node used on the operator selection algorithm

Operator, contains information about a single operator	
Op.name	Operator's name
Op.mem	Operator's memory usage
Op.node	Node at which the operator is deployed
Op.downOP	Operator's downstream peer (null if last operator)
Op.upOPs	Array of operator's upstream peers (null if first operator)
Transfer, contains the information for a possible set of operator that can be transferred	
Transfer.ops	Operators to be transferred together
Transfer.node	Destination node
Node, contains information about the operators deployed at a node and the available transfers	
Node.ops	Array of Operator deployed at the node
Node.transfers	Array of Available Transfer actions for the node

The operator selection algorithm makes use of three types of objects: Operator, Transfer and Node. Their classes are described on table 1. So each time a new operator is deployed to the node the algorithm on listing 1 is executed to calculate the possible combinations of operators transfers and save them to Available Transfer array.

```
1.  Forall (Operator op in node.ops)
2.     // Create an array containing this operator
3.     OP[] mustMoveTogether = {op}
4.     // Add to array mustMoveTogether downstream operators of op at the same node
5.    Node dst = getDownOPsInNode(mustMoveTogether, op.downOP, node)
6.     Transfer t(mustMoveTogether, dst);
7.     node.transfers.add(t);
8.     // Check if the operator has only one upstream peer
9.     If (allSingleUpstream(op,node)) {
10.       OP[]mustMoveTogether = {op}
11.       Node dst = getUpOPsInNode(mustMoveTogether, op.upOPs[0], node)
12.       Transfer t(mustMoveTogether, dst);
13.       node.transfers.add(t);
14.    }
15. }
```

Listing 1 – Algorithm for finding the possible combinations for operator transfers

On listing 1 we describe the operator selection algorithm which identifies all the possible Transfers that can be issued when a SPE node needs to transfer operators (for example if the node is overloaded). For all Operator op deployed at the node we check: a) if there are any downstream operators of the op in the same node (lines 2-7) and b) if there are any upstream operators of the op in the same node (lines 8-13). From this information, the algorithm creates and determines which Available Transfers are applicable to the parent node and which to the child(ren) node(s).

The algorithm begins by creating an array (mustMoveTogether) in which it saves the downstream operators of op (line 3). More specifically it uses faction getDownOPsInNode() to find the downstream operators of op that might be deployed at the same node and add them to the array mustMoveTogether (line 5). Remember that if op's downstream operators belong to the node, the only way to transfer op to the parent node is to create a Transfer that would transfer all of them to the parent node (section 2.2.2, second scenario). Next algorithm creates Transfer t, in which it adds the operators from array mustMoveTogether and sets the destination node (i.e. the parent node, line 6). Last the algorithm adds Transfer t in Available Transfer array (line 7).

The algorithm continues by calculating all the Available Transfers that would involve op and they would be applicable only to children nodes. For each operator we save the information related to all of its upstream operators, belonging to the same node on op.upOPs. The algorithm checks if all the operators in op.upOPs have only one input data stream, using the function allSingleUpstream() (line 9). We perform this check for the cases where there are two (or more) upstream data streams that are feeding the operator op (section 2.2.2 third scenario). In those cases the algorithm does not create a Transfer that involves the transfer of the specific operator to any child node; the LB protocol is only able to transfer the operator to the parent node.

If result from allSingleUpstream() is true, the algorithm creates an array (mustMoveTogether) in which it saves the upstream operators of op in the node (line 10). Faction getUpOPsInNode() adds the upstream operators of op that might be deployed at the same node, at the array mustMoveTogether (line 11). Next algorithm creates Transfer t, in which it adds the operators from array mustMoveTogether and sets the destination node (line 12). In this case the destination node, is the node that feeds (provide the input stream) the last operator in the array op.UpOps. Last the algorithm adds Transfer t in Available Transfer array (line 13).

```
1.  Node getDownOPsInNode (OP[] ops, OP op, Node node){
2.    If (op.downOP!=null && op.downOP.node==node){
3.      ops.add(op.downOP)
4.      getDownOPsInNode(ops,op.downOP,node)
5.    }
6.  }
7.  Boolean allSingleUpstream (Op op,Node node){
8.    If (op.upOps[0].operator.size() == 1){
9.      If (op.upOps[0].node == node){
10.       Return allSingleUpstream (op.upOps[0], node)
11.     }Else // Upstream operator of op doesn't belong to node{
12.       Return true
13.     }
14.   }Else // Operator op has more than one upstream peers{
15.     Return false
16.   }
17. }
18. Node getUpOPsInNode (OP[] ops, OP op, Node node){
19.   If (op.upOPs[0]!=null && op.upOPs[0].node==node){
20.     ops.add(op.upOPs[0])
21.     getUpOPsInNode (ops, op.upOps[0],node)
22.   }
23. }
```

Listing 2 – Pseudo code for the functions on listing 1

Listing 2 contains the pseudo-code for the three functions used on the operator selection algorithm on listing 1.

allSingleUpstream () function checks if operator op has only one upstream data stream. The function begins by checking the first entry in the array with the upstream operators of op. If op has only one upstream operator (i.e. there is only one input stream for this operator), line 8, the function continues checking all the entries on op.upOPs until it finds the last operator in the node. The last operator in the node is the one that has an upstream operator that is deployed on a different node (lines 9 - 12).

3.3 Protocol detailed description

With the Load Balancing protocol we want to provide load balancing among the nodes. In order to measure the performance of a node we check the average memory utilization at a specific time point. We could also measure CPU usage or a combination of CPU and memory measuring together. But for the purposes of this project we focus on memory usage only. According to the protocol, node decides if it is necessary to transfer (or receive) operators to a direct adjacent node (i.e. parent node or any of its children nodes).

More specifically each node of the system checks with period T, what is the memory usage of the SPE node. The node can be overloaded, underloaded and working normally. If a node works normally, the node is not undergoing any reconfiguration action. If the node is overloaded, it begins a negotiation round with its direct adjacent nodes in order to transfer to them operators. If the node is underloaded, it begins a negotiation round with its direct adjacent nodes in order to take on (transfer) operators from them.

When a node begins the negation for transfer of operators with its adjacent neighbors, the node must finish negotiations that it has already began, and don't begin new negotiations until then.

The figure 11 represents a bar with the total memory utilization of a node. The protocol is using three thresholds: Lower threshold (L_t), Target threshold (T_t) and Upper threshold (U_t), which we have noted on the memory bar. According to them we define if a node is underloaded, working normally, or overloaded.

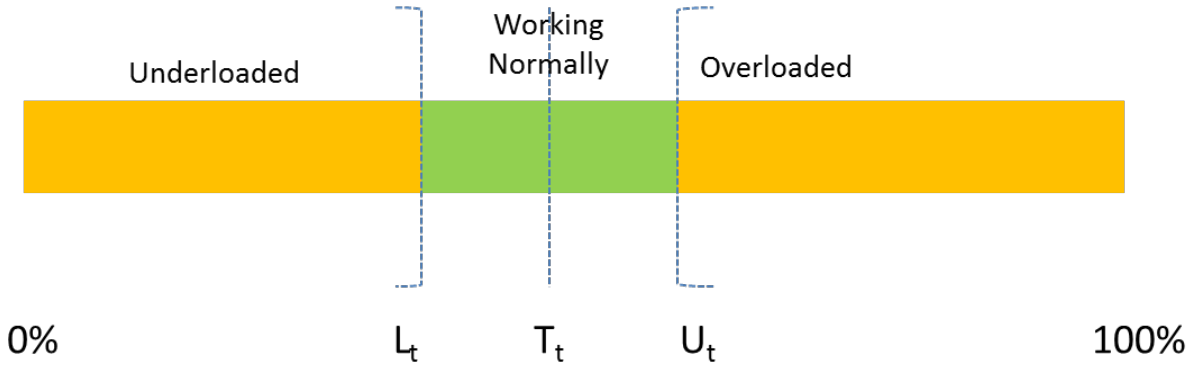

Figure 11- Memory utilization of a SPE node

- Node is underloaded when the memory usage of it is below the Lower threshold (L_t) point.
- Node is overloaded when the memory usage is above the Upper threshold (U_t) point.
- Node is working normally when the memory usage is between the L_t and U_t points.

When a node is transferring operators it aims to reach the Target threshold (T_t) point.

T_t is the average value between Lower and Upper thresholds and is chosen instead of U_t to avoid situations where nodes are constantly transferring operators; if we would transfer an operator to a node that is working normally but its memory usage is very close to U_t (for example the memory usage of the node after the operator transfer gets 1% less than the U_t) there is a high probability that this node would run out of memory soon. This would initiate another execution of the protocol and new round of operators transfer.

Table 2 – Protocol message

Message, information message used by the Load Balancing Protocol	
Message.sender	Node which sent this message
Message.typeOfMessage	The type of the message
Message.urgent	Used by the protocol to sort messages of the same type (Values : High, Normal, Unset)
Message.size	The total memory size of operators present in the Message.transfers
Message.transfers	Array of available Transfer actions carried in the message

Table 2 presents the structure of the messages that we use with the LB protocol. The types of messages in the protocol can be divided in two groups; whether the node that initiates the negotiation is overloaded or underloaded (table 3).

Table 3 – Types of the protocol messages

Type of message	
Types related to the negotiations of an overloaded node	opOfferRequest
	opOfferReply
	opOfferTransferRequest
	Clear
	opOfferTransferReply
Types related to the negotiations of an underloaded node	opDemandRequest
	opDemandReply
	Clear
	opDemandTransferRequest

In the protocol the node that initiated the negotiation can receive multiple messages (one for each adjacent node) of one of two types “opOfferReply”, “opDemandReply”. Message.urgent is used for messages of these two types and is used to sort messages of the same type. For example a node receives two “opDemandReply“ messages, one from an overloaded node and one from a normal operating node. The message from the overloaded node would be sorted to be processed first in the protocol.

In figure 12 we present an example of the negotiation for operator transfer between an overloaded node and its adjacent nodes. We are using 3 nodes, node 2 where the operator is initially deployed, its parent node 1 and child node 3 (figure 12). We examine node 2. Node 2 is overloaded because node’s memory usage is 65%, 5% above the U_t (thresholds are specified in the figure). Node 2 has a number of different operators deployed; among them is operator A (memory size consumed by the operators are specified in the figure). Node’s 1 memory usage is at 59%. Node’s 3 memory usage is 40%. Nodes 1 and 3 have their own distinct group of operators. In this scenario these are the steps that our LB protocol follows:

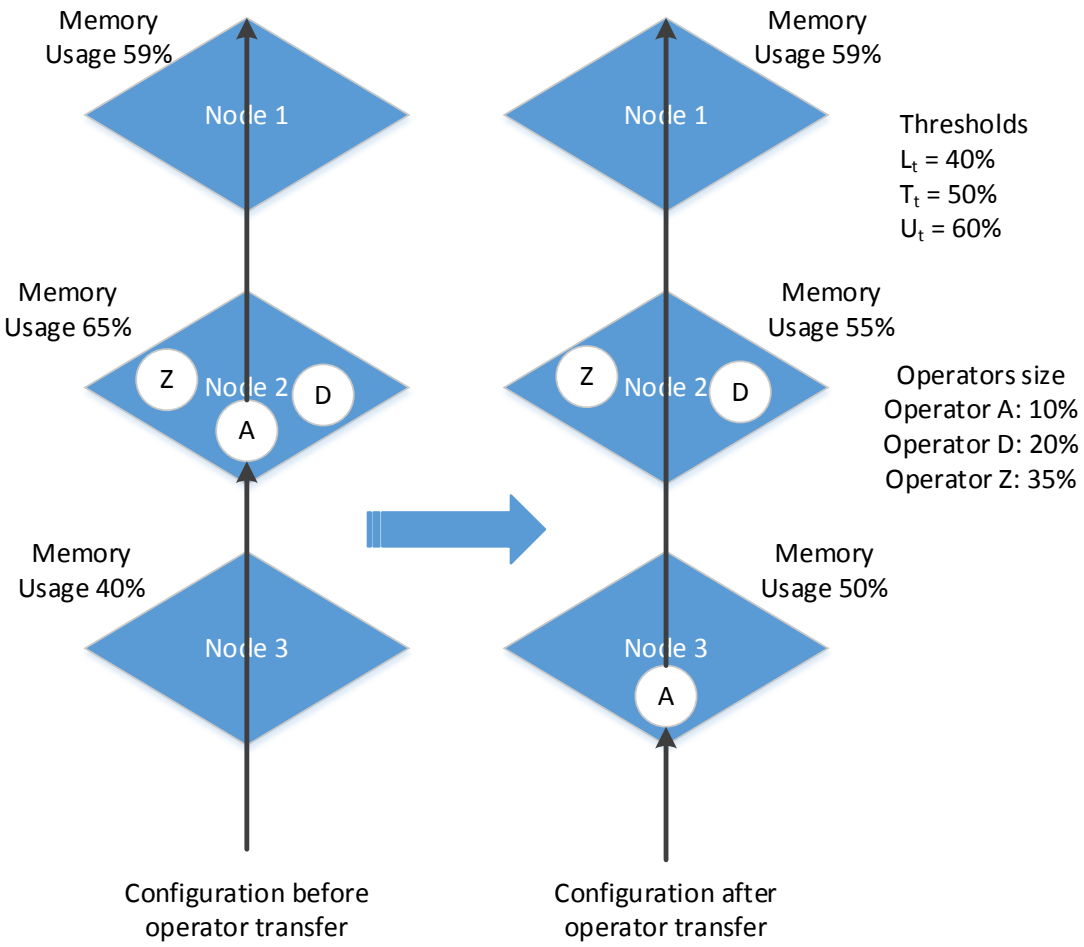

Figure 12 – Operator transfer because node 2 is overloaded

- Phase 1. Node 1 exchanges information messages with the adjacent nodes – sends "opOfferRequest" message to every adjacent node – stating that it needs to transfer operators. Node 2 would first check its available transfers and would look for which operator can transfer in order to reduce its memory usage to T_t. Node 2 would choose to send available transfers that contain operator A to the other nodes, because it's the only operator that can transfer and not reduce the node's memory usage bellow T_t.

- Phase 2. The adjacent nodes examine "opOfferRequest" message and decide on if they are able to receive operator A from node 2. More specifically each node examines in which state it is going to get in (normal or overloaded), if they receive the operator offer from node 2. Then each node sends an "opOfferReply" message as a reply to node 2.

- Phase 3. Node 2 receives the replies from the adjacent nodes. Node's 1 reply states that it cannot take on the operator A (if node 1 takeover operator A the node would get overloaded). Node's 3 reply state that the node can take over operator A with size 10%. Node 2 informs node 3 that is going to transfer operator A with message "opOfferTransferRequest". Node 2 sends also clear message to Node 1.

- Phase 4. Node 3 loads the operator 1 in its main memory and confirms the decision of node 2 with "opOfferTransferReply" message.

Figure 13 shows the sequence of messages exchanged on the previous example and in general when we transfer operators because a node is overloaded.

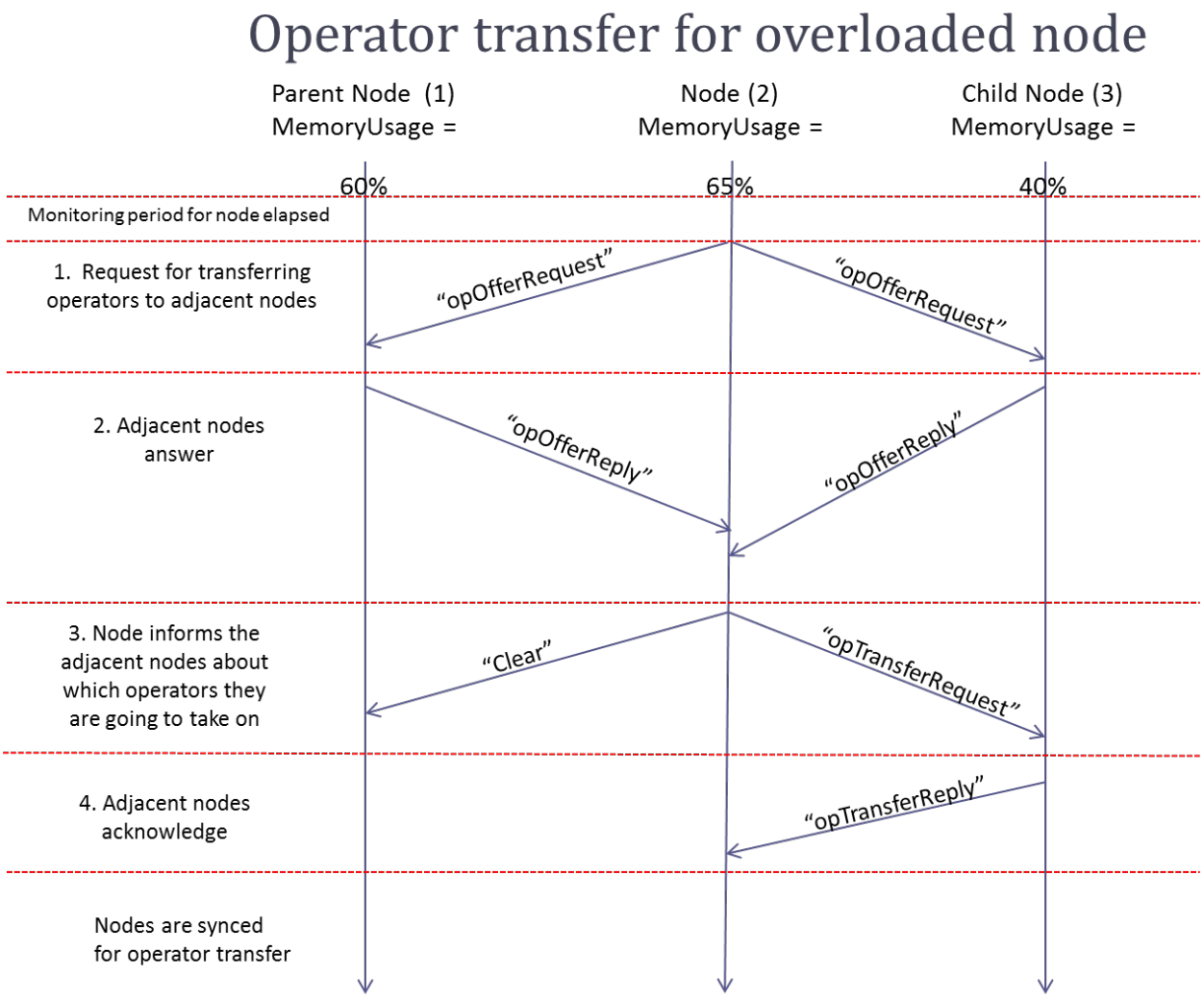

Figure 13– Sequence of messages exchanged when a node is overloaded

Listing 3 shows the pseudo code common to all nodes. Code is executed when the monitoring period for the node has elapsed. If memory usage is above the U_t, the node decides which operators should be transferred to another node (lines 1 – 16). If the node has memory usage below L_t the pseudo code in lines 17 -21 is executed instead.

Upon: new monitoring period has elapsed at the node

```
1.  If [(Mem_av > U_t ) && (No negotiation in progress)] Then{
2.      Message parentMessage(node,"opOfferRequest", Unset,0, Transfer[] empty)
3.      Message childMessage(node,"opOfferRequest", Unset,0, Transfer[] empty)
4.      excessOpSize := Mem_av –T_t // excessOpSize variable indicates the total memory size of the operators
        that need to be moved
5.      For (Transfer t from node.transfers){
6.          If (t.totalOpMem()  <= excessOpSize)
7.              If (t.node == Parent_Node) {
8.                  parentMessage.transfers.add(t)
9.              }
10.             Else{
11.                 childMessage.transfers.add(t)
12.             }
13.     }
14.     Send parentMessage   to      parent_Node
15.     Send childMessage    to      child_Node
16. }
17. Else If [(Memav. <L_t) && (No negotiation in progress)] Then{
18.     Message opRequest(node,"opDemandRequest",Unset,0,Null)
19.     opRequest.size := (T_t – Mem_av)
20.     Send opRequest  to all adjacent Nodes
21. }
```

Listing 3

If the node is overloaded it creates "opOfferRequest" messages that correspond to each of the node's adjacent neighbors (lines 2-3). Node sets in local variable *excessOpSize* the total memory size of the operators that should be transferred to other nodes (line 4). Then all available Transfers *t* available in the node.transfers that are smaller in size than excessOpSize are selected (lines 5-6). A copy of transfer *t* is added to the message that corresponds to t.node (lines 7-13). When the node has finished checking all the available Transfers, it sends the "opOfferRequest" messages (lines 14-15).

If the node is underloaded, it creates an "opDemandRequest" message (line 18). Node sets the *opRequest.size* field to $T_t – Mem_{av}$, because this is the memory size of the operators that the node should deploy to reach T_t (line 19). Last a copy of the "opDemandRequest" message is sent to all the direct adjacent nodes.

Upon: receive Message *mess,* with field type = "opOfferRequest" and *source* the Node that sent the *mess*

```
1.  Message replyMessage(node,"opOfferReply",Unset,0, Transfer[] empty)
2.  If (Mem_av > U_t) then
3.          Send replyMessage to source Node // the node is Overloaded that's why it can't receive more
    operators. For this reason node reply with an empty  message
4.  Else{
5.          target := U_t - Mem_av - 1
6.          For (Transfer t from mess.transfers){
7.              If ( replyMessage.size <= target){
8.                  If (t.totalOpMem()  + replyMessage.Size <= target)
9.                      replyMessage.transfers.add(t)
10.                     Increase replyMessage.size by  t.totalOpMem()
11.                     mess.transfers.remove(t)
12.                 }Else
13.                     Break For Loop
14.             }
15.         If (Mem_av <= L_t)
16.           replyMessage.urgent := High
17.         Else
18.           replyMessage.urgent := Normal
19.         Send replyMessage to source Node
20. }
```

Listing 4

Listing 4 presents the pseudo code executed when a node receives an "opOfferRequest" message (*mess*). Node begins by creating an "opOfferReply" message (*replyMessage* message-line 1). If the total memory usage of the node is above U_t (line 2), the node is not able to deploy more operators. For this reason the node just sends the reply message with parameter *replyMessage*.urgent = Unset, *replyMessage*.size = 0 and *replyMessage*.transfers = empty Transfer [], which according to the protocol indicates that the node cannot take on any operators (line 3).

If node's total memory usage is below U_t the node continues and sets the local variable *target* equal to the size of memory that the node can use to deploy more operators (line 5). A node can accept operators as far as they do not exceed the upper threshold. A more conservative approach would be to accept operators as long as they do not exceed the target threshold. In our case we go for the first option and we set *target* equal to (Ut - Memav – 1).

Next node checks the Transfer *t* that is present on the *mess*.transfers. *replyMessage*.size is equal to the total memory size of the operators that the node has decided to take on (initially set to zero). If the reply *replyMessage*.size is smaller than *target* (i.e. the node can take on more operators) the node checks if there are more *t* in the *mess*. The node continues with the check on line 8. At this point the node checks if operators at *t* can be loaded in the node. This is done by calculating the overall size of *t* (t.totalOpMem() function returns the total memory usage of

the operators present in *t*). If operators in *t* can be deployed on the node, *t* is added on the reply message and we increase the *replyMessage.size* by t.totalOpMem*()* (lines 9 - 10). When a node selects a transfer *t*, it decides the operators it can receive. For this reason the algorithm removes from *mess* the rest of the transfers that involve the operators that are present on *t* (line 11). The node checks the *t* present in *mess* until it cannot receive more operators.

Next node sets *replyMessage.urgent* according to its memory usage (lines 15 - 18). Reply messages from nodes with total memory usage below L_t have greater priority in this phase of our protocol. For example, in case that there are two nodes that can take over the same operators, one node underloaded and one normally working; we want underloaded node to take over the operators. For this reason we set *replyMessage.urget* to high.

Upon: receive Message(s) *mess*, with field type = "opOfferReply"

```
1.  repliesList.add(mess)
2.  if (node received all the reply message from the adjacent node){
3.     Set agreedSize := 0 // agreedSize is a local variable on the node, that indicates the size of the operators that  it is agreed to be transferred to another node.
4.     For (each mess in the repliesList){
5.        Message replyMessage(node,"Clear", Unset,0, new Transfer[] empty )
6.           If (agreedSize < excessOpSize){ // excessOpSize is a local variable on the node which was set on listing 3 at line 4
7.              For (Transfer t from mess.transfers)
8.                 If (t available in node.transfers){ // the node checks that if the Transfer that the adjacent node want to perform is still available in the list of available transfer in the node
9.                    If (t.totalOpMem()  <= excessOpSize - agreedSize){
10.                      Increase agreedSize by t.totalOpMem()
11.                      replyMessage.transfers.add(t)
12.                      Increase replyMessage.size by t.totalOpMem()
13.                      replyMessage.typeOfMesage := "opOfferTransferRequest"
14.                      node.transfers.remove(t)
15.                   }//End If
16.                }//End If
17.             }//End For Loop
18.             Send replyMessage to mess.sender // send a message which contains the actions that the adjacent node will perform
19.             }
20.          }Else
21.             Send replyMessage to mess.sender // send an empty message that indicates to the adjacent node that it will not do any action and the negation has finished for him.
22.    }//End for
23.    Clear RepliesList
24.  }//end If
```

Listing 5

Listing 5 presents the pseudo code executed by the node that began the negotiation (overloaded node) and is at the phase where it collects the replies from the adjacent nodes. So the node that initially sent the "opOfferRequest" messages to its adjacent nodes, collects the "opOfferReply" messages from them. Every "opOfferReply" message that the node receives is stored it at repliesList array. Messages are sorted in the list according to the message.urgent field. The node waits until it receives all the reply messages and only then continues to process those (lines 1-2).

The node checks the Transfers *t* that are present in each "opOfferReply" reply message. If the node has not reached its goal, i.e. *excessOpSize*(line 6),the node checks if *t* is still available in node.transfers (line 8). This check is performed in order not to assign the same *t* to two different nodes.

Upon: receive Message *mess*, with field type = "opOfferTransferRequest" and *source* the Node that sent the Message

1. Message *transferReplyMessage* (node,"opOfferTransferReply",Unset,0, **new** Transfer**[] empty**)
2. **For** (Transfer t from mess.transfers){
3. node.ops.add(t.ops) // load the operators that are present in the Transfer t to the main memory of the node and add them to the list with the operators running in the node
4. *transferReplyMessage*.transfers.add(t) // indicate which actions are performed by the node
5. **Increase** *transferReplyMessage*.size by t.totalOpMem()
6. }//**End For Loop**
7. **Send** *transferReplyMessage* **to** *source*

Listing 6

Listing 6 presents the pseudo code executed on the node receiving "opOfferTransferRequest" message (*mess*). The node loads the operators indicated on each Transfer *t* present on *mess*. On the same time the node adds *t* on the reply message *transferReplyMessage.* *transferReplyMessage* is used as an acknowledge message that indicates that the node has successfully deployed the operators.

Upon: receive Message *mess*, with field type = "opOfferTransferReply"

1. **For (**Transfer t from mess.transfers**)**
2. node.ops.remove(t.ops) // node removes the operator from the list of running operators and begin to forward the tuples related to this operator

Listing 7

Listing 7 presents the pseudo code executed on the node that receives an "opOfferTransferReply" message (i.e. the node that was overloaded and initiated the negotiation process). The node removes from main memory the operators present on *t*. After this point the node only forwards data streams that are related to the operators that are removed from main memory (*t.ops*).

Figure 14 shows the sequence of messages exchanged when a node announces that can deploy more operators because it is underloaded.

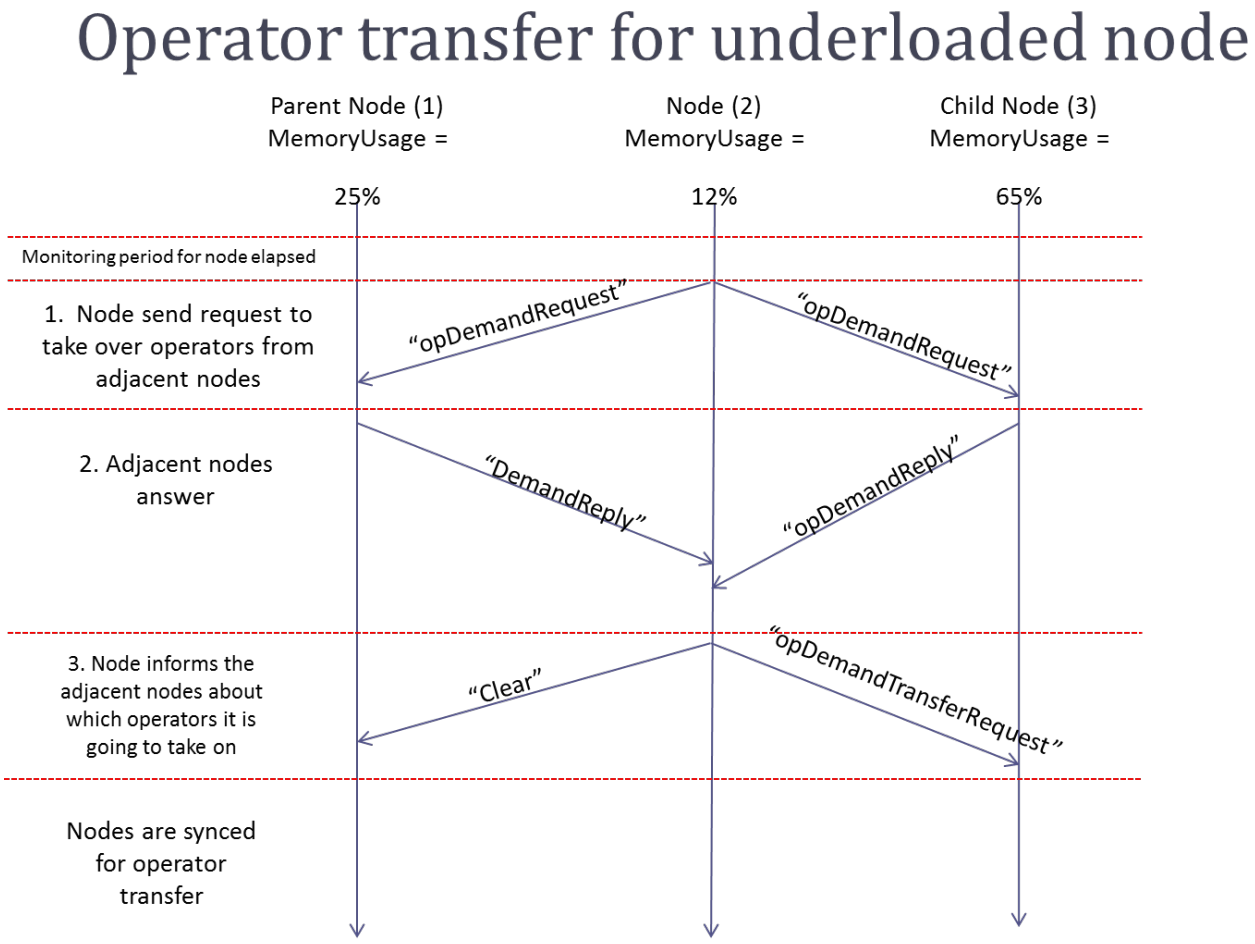

Figure 14 - Sequence of messages exchanged when a node is underloaded

Listings 8 - 10 provide the pseudo code that is executed by the LB protocol when there is negotiation process which a node initiated because the node was underloaded.

At listing 8 we can see the pseudo code that each node that receive message (*mess*) of type "opDemandRequest" executes. If node has memory usage equal or smaller than T_t, it cannot provide any operators and for this reason the node replies with an empty "opDemandReply" message (lines 2-3). If the node has memory usage above T_t it checks the available Transfers *t* that are in the node (*node.transfers*) and the ones that are smaller or equal to *target* are added on the reply message (lines 4 - 16). Next node is setting the priority that the reply message would have (lines 17 - 20) and sends the message to *source* node.

Upon: receive Message *mess*, with field type = "opDemandRequest" and *source* the Node that sent the *Message*

```
1.  Message  replyMessage(node,"opDemandReply",Unset,0, new Transfer[] empty)
2.  If (Mem_av < T_t) then
3.          Send replyMessage to source Node  // the node cannot offer any operators and reply with an
    empty message
4.  Else{
5.          If ((Mem_av - mess.size) < T_t)
6.            target := Mem_av – T_t  // target is a local variable that indicates the memory size of operators
    that the node could offer
7.          Else
8.            target := mess.size
9.          For (Transfer t available in node.transfers){  // check all the available Transfers on the node
10.           If (t.node == mess.sender)  // if Transfer is related with the node that sent the mess
11.             If (t.totalOpMem()  <= target){
12.               replyMessage. transfers.add(t)
13.               Increase replyMessage.size by t.totalOpMem()
14.             }//End If
15.           }//End If
16.         }// End For Loop
17.         If (Mem_av <U_t)
18.           replyMessage.urgent := Normal
19.         Else
20.           replyMessage.urgent := High
21.         Send replyMessage to source Node
22. }
```

While executing: all other received messages ignored

Listing 8

Listing 9 presents the pseudo code executed on the node that initially sent the "opDemandRequest" messages (the underloaded node, listing 3) to its adjacent nodes and then collects the "opDemandReply" messages from them. Every "opDemandReply" message that the node receives is stored at repliesList array. Messages are sorted in the list according to the message.urgent field. The node waits until it receives all the reply messages and only then continues to process them (lines 1-2).

If the node received an empty "opDemandReply" message (i.e. *mess.size* ==0 that the adjacent node *mess.sender* sent), the node sends a "Clear" message to *mess.sender* and the negotiation with that node finishes (lines 8 -9). Otherwise the node chooses *t* and loads the operators present at *t.ops* to *node.ops*(lines 12-13). The node also adds the *t* in the "*opDemandTransferRequest"* reply message (lines 14 -16). When a node selects a *t* it removes all other *t* present on mess.transfers that have common operators (for example two Transfer *t* that have common operators in *t.ops* – line 18).

Upon: receive Message(s) *mess*, with field *type* = "opDemandReply"

```
1.  Add mess to repliesList
2.  if (node received all the reply message from the adjacent node){
3.     agreedSize := 0 // agreedSize is a local variable on the node, that indicates the size of the operators that the local node has agreed to take over from the adjacent nodes
4.     target := T_t - Mem_av // target is a local variable that indicates the memory size of operators that the node could take over
5.     For (each mess in the repliesList){
6.       Message  trasferRequest(node,"Clear",Unset,0, new Transfer[] empty)
7.       If (agreedSize <= target){
8.          If (mess.size==0)  // if the adjacent node replied that it cannot offer any operators (mess.Size==0) the node replies with a clear message and the negotiation finishes for the adjacent node
9.             Send trasferRequest to mess.sender
10.         Else{
11.            For (Transfer t from mess.transfers){
12.               If (t.totalOpMem()  <= (target – agreedSize)){
13.                  node.ops.add(t.ops) // load the operators that are present in the transfer to the main memory of the node and add them to the list with the operators running in the node
14.                transferRequest.transfers.add(t)
15.                Increase  transferRequest.size by t.totalOpMem()
16.                transferRequest.typeOfMesage :="opDemandTransferRequest"
17.                Increase agreedSize by t.totalOpMem()
18.                 node.transfers.remove(t)
19.              } // End If
20.           }//End For Loop
21.           Send  trasferRequest to mess.sender  // the reply message with all the actions that the node is going to (i.e. the operators that is going to takeover)
22.        }// End Else
23.     }Else
24.        Send trasferRequest to mess.sender // the adjacent node replied with a set of actions but the node won't perform them. The node replies with a clear message and the negotiation finishes for the adjacent node
25. }//End For Loop
26. Clear RepliesList
```

Listing 9

Listing 10 presents the pseudo code executed on the node that receives an "opDemandTransferRequest" message. The node removes from main memory the operators present on *t*. After this point the node only forwards data streams that are related to the operators that were removed from main memory (t.ops).

Upon: receive Message *mess*, with field *type* = "opDemandTransferRequest"

```
1.  For (Transfer t from mess.transfers){
2.     node.ops.remove(t.ops) // node removes the operator from the list of running operators and begin to forward the tuples related to this operator
3.  }
```

Listing 10

4. Evaluation

4.1 Evaluation Setup

In order to evaluate the performance of the LB protocol described in the previous section, we used NETSIM 5.2 program [12] to simulate a network of nodes. Each node we assume that can run a stream process application. We used a dual-core Intel® Core™i5 CPU 450M@ 2.40GHz computer equipped with 4GB of RAM to run all the tests. The protocol and all the testing code are written in Java 1.8.0-ea-b108 using the default classes provided by the NETSIM program.

We want to simulate a hierarchical network of devices, so we created a 15 node perfect binary tree, with bidirectional communication lines among the adjacent nodes. Each node in the simulation represents a node in the AMI communication network, which we assume that it can run a distributed SPE application (SPE node). All nodes have equal computational resource and are running the same version of the LB protocol.

Our purpose is to measure how our protocol performs when there are fluctuations on the creation rate of the data stream. For this reason we simulated an environment that has sudden changes at the memory usage of each operator. To achieve this we created a component that modifies the operators' memory usage at runtime. We artificially increase or decrease the memory usage of an operator to replicate the sudden changes in the rate that tuples arrive.

Table 4 – Parameters of experiments

Parameters of an experiment	
Number of operators	The total number of operators that are deployed to all the nodes
Monitoring Period	Period with which each node checks its memory usage and runs the LB protocol if needed
Timespan (sec)	The time interval that we choose a new operator to alter its memory usage. Also equal to the length of time that we alter the memory usage of this operator.
Period (sec)	The period we change the operator's value that represent its memory usage size
Increase Rate	Random value taken from a Poisson distribution with mean (λ) equal to 2

Table 4 lists the parameters that we set for every simulation run.

Each run of the simulation lasts for 16 minutes. The simulation begins with all nodes loading the operators and after 2 seconds we begin to alter the memory size of the operators in the system.

More specifically, every *Timespan* seconds we choose an operator (the selection of the operator is done following a discrete uniform distribution of all the operators in the system). For this timespan we command the node that possesses this operator to increase/ decrease the operator's memory size every *Period* seconds for the *Increase Rate* value. From the beginning of the simulation and for the next 8 minutes the operators are only increased. On the 8th minute of the simulation and for the next 8 minutes operators are only decreased. Last, in our implementation we only decrease the memory usage of an operator that we have previously increased.

Next we present a general example on how we alter the memory size of an operator. We assume that this example takes place in the first 8 minutes of the simulation run and therefore we only increase the value of the operator. We command node 5 that for the next 20 seconds (Timespan) the node will increase every 2 seconds (Period) the memory size of operator Y, by a different value that it is taken from Increase Rate. Every 5 seconds (Monitoring Period) each node of the system is running the LB protocol and performs accordingly.

4.2 Evaluation of Results

In this section we evaluate our LB protocol on simulated environment. We examine how a distributed SPE application installed on the AMI communication network performs with and without the LB protocol.

In both cases, we measure the average memory usage for all nodes and the standard deviation. In order to achieve that, we are logging every two seconds the total memory usage of each node. Then for every two second we calculate how much is the average memory usage of all nodes. Also every two seconds we calculate the standard deviation. What we want to do is to study the variation of the total memory usage of each node compared to the average memory usage of all nodes.

For the evaluation of the protocol we compare pairs of simulation runs: one simulation run in which the nodes don't execute the LPB and one simulation run in where the LBP is executed. In both simulations runs we have the same distribution of operators per node, i.e. each time, when the simulations are initialized, a node will possess exactly the same operators. Also, in both simulations we increase the same operators, on the same sequence and with the same parameters for each operator (i.e. increase/decrease, Timespan, Period and Increase Value per operator).

We proceed with a detailed overview of one experiment, where we evaluate the results that we receive for that experiment. Then we present summary results for all the experiment runs.

On table 5 we list the parameters of one simulation. Each simulation run begins by randomly deploying 360 operators on the 15 nodes of our situation environment. Then after 2 seconds, we randomly peek on of the 360 operators. For the next 16 seconds and every 1 second we increase its memory size by a random value by that it is taken from Increase Rate. When the 16 seconds pass, we chose another operator and begin to increase its memory size accordingly.

Table 5 – Simulation example parameters

Parameters of experiment	
Number of operators	360
Monitoring Period (sec)	5
Timespan (sec)	16
Period (sec)	1
Increase Rate	Poisson distribution with mean (λ) equal to 2
Threshold values (percentage of nodes main memory)	L_t = 40 % T_t = 50 % U_t = 60 %

On figure 15 we see two graphs with Average Memory Usage for all nodes and standard deviation, and how they change over time. On the first graph we can see that the average memory usage, the first 8 minutes we have a constant increase and for the next 8 minutes it decreases. This has to do with the fact that the first 8 minutes we are randomly pick operators and we increase their memory size, while on the next 8 minutes we decrease the memory size of the operators that we previously increased.

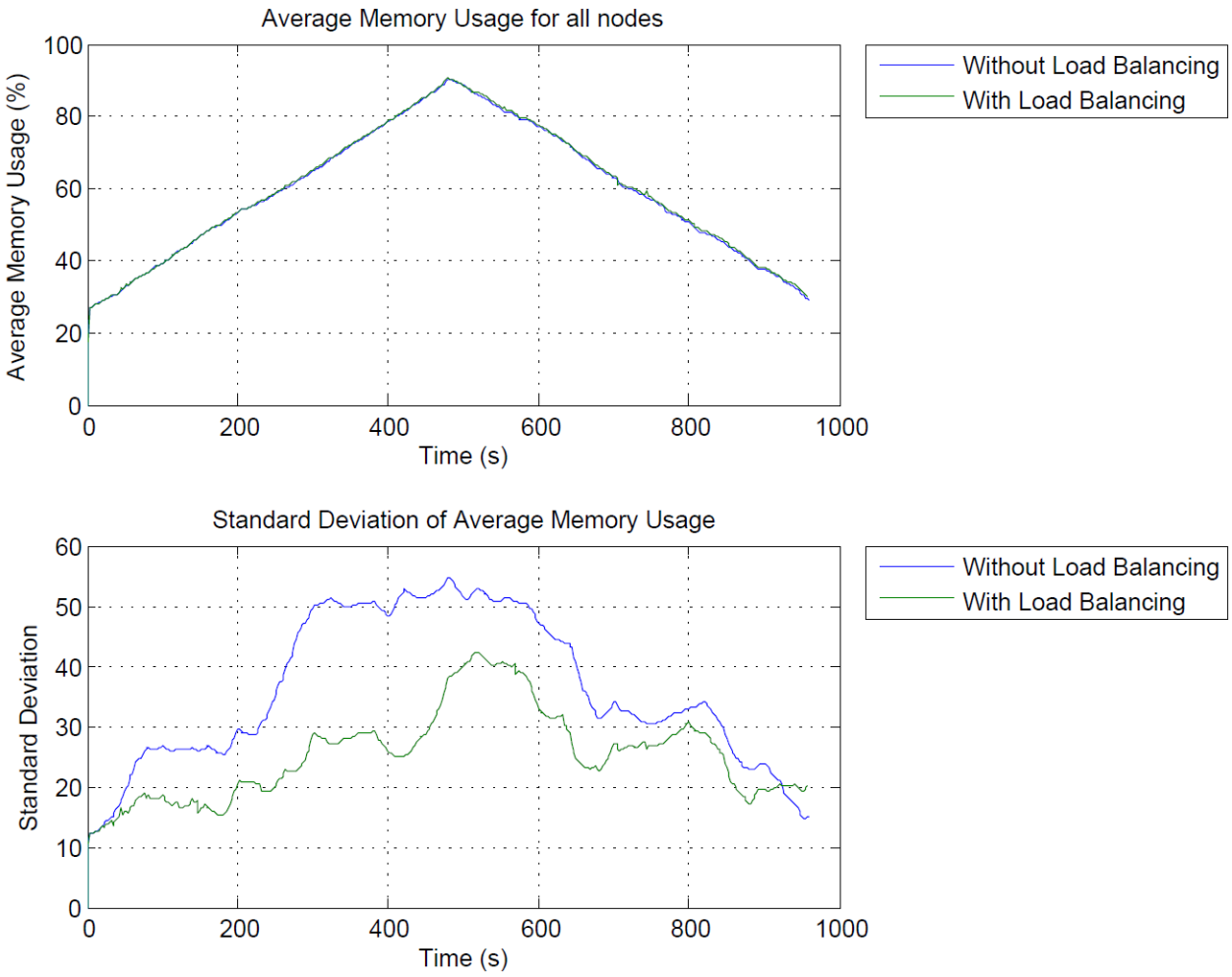

Figure 15 – Average Memory Usage and Standard Deviation

On the second graph in figure 15, we see how standard deviation of average memory usage changes over time. As we observe system in which we have enabled the LB protocol has lower standard deviation values during the execution of the simulation run. Nodes' memory usages have smaller variation from the average memory usage for all nodes when we used the LB protocol. We manage to achieve more even distribution of the workload among the nodes of the system, as we see from the standard deviation graph. This fact can be interpreted to more even distribution of the workload.

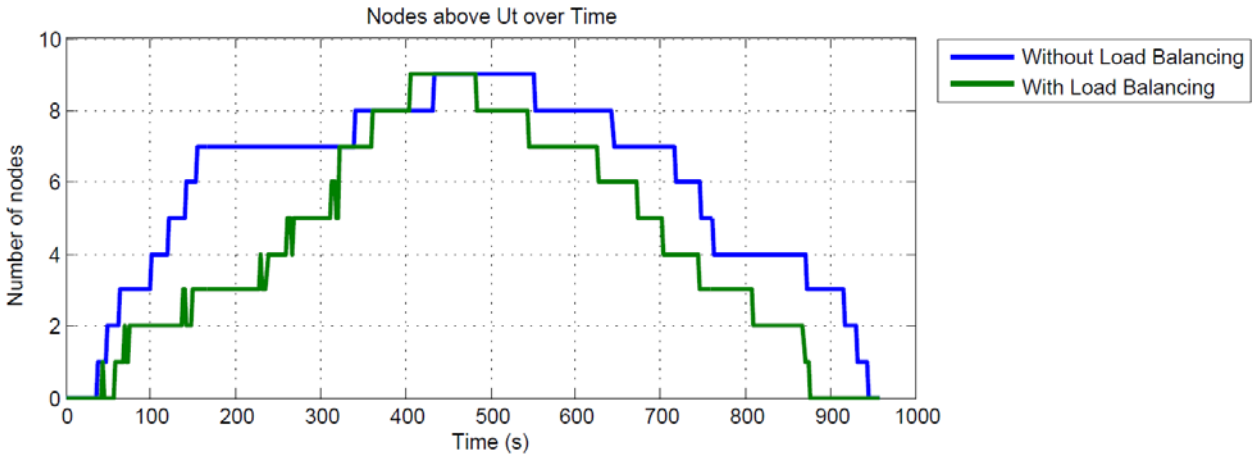

Figure 16 – Number of nodes above U_t over time

All the nodes in the simulations begin either underloaded or normally working. In the simulations we are monitoring when a node will get overloaded and for how much time is going to stay in that state.

On figure 16 we see how the number of overloaded nodes changes over time. Here we observe that when we use the LB protocol the system has less overloaded nodes over time. Also the nodes that get overloaded stay on that state for shorter time. We calculated that when we use the LB protocol the same system processes the same workload being 26,44 % less overloaded in total. In addition the number of the nodes that get overloaded is smaller in 85,52 % of the time.

On average in a series of 15 experiments with similar parameters, at the simulation runs that we used the LB protocol, the protocol kept the number of the overloaded nodes smaller in 80,83 % of the time.

Overall, the results show that with the use of LB protocol we can minimize the time that the nodes in a simulated distributed SPE system are overloaded due to the fluctuations of the creation rate of the data streams. We can observe that nodes of the system can tolerate the sudden increases of workload because for the same amount of workload there are less overloaded nodes in most cases.

5. Related Work

There have been different approaches on developing load balancing mechanisms that would improve the performance of Distributed Stream Processing Engines.

Flexible Filters [13] is a load balancing solution applicable to multi-core processor architectures and using backpressure messages. More specifically multiple copies of data processing filters are deployed simultaneously on multiple cores for a short amount of time. The idea is that data from the same data stream are split in two separate streams and are processed in parallel by two cores on the same time. Each core knows in advance the maximum throughput of the filters deployed on it. When the throughput of a filter falls (because the core cannot keep up with the processing rate of the upstream core), the core sends backpressure signals to upstream core. The upstream core deploys the filter related to the backpressure signal and processes in parallel part of the data (that belong to the data stream that the other core cannot process on time). Such a solution cannot be directly applicable in our distributed SPE system. In our system there are operators that have more than one input data stream. In a case where an operator has two input streams that originate from two different child nodes, we would not be able to deploy two copies of the operator at the two child nodes and process the data in parallel, since each node has access to only one data stream.

In bounded-price mechanism [14] nodes of the system conduct two-way contracts among them. Nodes negotiate off line with their neighbors about the workload that they can take on in return of payment. Each node tries to achieve the best economic deal taking into consideration the workload that must be processed and the contracts that it has made. When a node needs to process data from a data stream, it checks if it is cheaper to process the data locally or "outsource" the processing to another node. On Stream Cloud (SC) [15]the system is divided to share-nothing clusters. In order to counter the variation in the workload, each cluster has the ability to commission/decommission new nodes in the cluster (elasticity) and also perform load balancing between the already existing nodes. For the load balancing part, SC monitors the standard deviation of CPU utilization. When the CPU usage standard deviation is above a predefined upper threshold, a new distribution of operators (among the existing nodes) is calculated. If with the new distribution of the operators the performance of the system improves, operators from the overloaded node are transferred to the node with the least workload. These two solutions, plus [16] could not be directly applicable on the system we examine in this thesis. These solutions have been designed for flat distributed SPE systems, where we can reroute data stream to any SPE node. The network we examine is hierarchical. As we presented, a data stream flows only from child to parent node and this raises limitations on operator transfer procedure, which these solutions don't examine.

6. Conclusion

The electricity utility companies in order to provide services like better management of the electricity network and dynamic price rates to the end users, need to have access to the information provided by AMI system in close to real time. These services would require to change over time the frequency that smart meters in the AMI system report their readings. Distributed SPE applications would be one of the solutions to process the data from the AMI systems, since they can process huge volume of data in close to real time margins.

In this thesis we developed a distributed Load Balancing Protocol to cope with the workload variations in a distributed SPE application installed on AMI system. Our goal is to avoid overloading and spread the workload between the SPE nodes of such distributed SPE system. This would require to transfer query operators between the SPE nodes. The distributed SPE application that we examined on this thesis, we assumed that is installed on a hierarchical AMI communication network and the data are always routed through the same nodes. For this reason we had the challenges: a) to transfer an operator to an adjacent SPE node that both have access to the appropriate data streams and b) have on the same time the available resources (free main memory). We developed an Operator Selection Algorithm that defines which operators can be transferred from one node to each of its neighbors (i.e. which actions are available for the node to perform). Each SPE node then uses the LB protocol to negotiate with its direct neighbors.

We developed the LB protocol that under simulated environment achieves the distribution of work load between SPE nodes of our system. We managed to decrease the number of the nodes and the time that nodes in a distributed SPE system stay overloaded, which we assume that it can be translated to better resource utilization in a distributed SPE application installed on AMI communication network. It is a subject for further research to verify that our LB protocol can bring the same results under a real distributed SPE system.

For the purposes of this report we measured performance according to nodes' main memory usage. Also all the decisions in the protocol are made based on nodes' memory usage. Although this is a good starting point, more parameters like nodes' CPU and Network utilization could be added to algorithm in a future work.

7. References

[1] M. Henzinger, P. Raghavan och S. Rajagopalan, ”Computing on data streams,” i *External Memory Algorithms: DIMACS Workshop External Memory and Visualization, May 20-22, 1998*, Palo Alto, California, American Mathematical Soc., 1998, p. 107.

[2] M. Stonebraker, U. Çetintemel and S. Zdonik, "The 8 Requirements of Real-Time Stream Processing," *SIGMOD Rec.,* vol. 34, no. 4, pp. 42-47, December 2005.

[3] D. Terry, D. Goldberg, D. Nichols och B. Oki., ”Continuous Queries over Append-only Databases,” i *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, vol. 21, New York, NY, USA, ACM, 1992, pp. Pages 321-330.

[4] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul och S. Zdonik, ”Aurora: a new model and architecture for data stream management,” *The VLDB Journal—The International Journal on Very Large Data Bases,* vol. 2, nr 2, pp. 120-139, August 2003.

[5] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. I. K. Datar, R. Motwani, U. Srivastava och J. Widom, ”STREAM: The Stanford Data Stream Management System,” *Book chapter*, Stanford InfoLab, 2004.

[6] S. Chandrasekaran, O. Cooper, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss och M. Shah, ”Telegraphcq: Continuous Dataflow Processing for an Uncertain World,” i *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, San Diego, California: ACM, 2003, pp. 668-668.

[7] D. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M.Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E.Ryvkina, N. Tatbul, Y. Xing och S. Zdonik, ”The Design of the Borealis Stream Processing Engine,” Asilomar, CA, Second Biennial Conference on Innovative Data Systems Research (CIDR 2005), 2005, pp. 277--289.

[8] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez och P. Valduriez, ”StreamCloud - A Large Scale Data Streaming System,” i *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on*, IEEE, 2010, pp. 126--137.

[9] ”The Future of the Electric Grid: An Interdisciplinary MIT study,” Massachusetts Institute of Technology, Cambridge, 2011.

[10] C. Bennett och D. Highfill, ”Networking AMI Smart Meters,” i *Energy 2030 Conference, 2008. ENERGY 2008. IEEE*, IEEE, 2008, pp. 1-8.

[11] J. Wang och V. Leung, ”A survey of technical requirements and consumer application standards for IP-based smart grid AMI network,” i *Information Networking (ICOIN), 2011 International Conference on*, IEEE, 2011, pp. 114-119.

[12] S.-A. Andréasson. [Online]. Available: http://www.cse.chalmers.se/~andreass/champ/Netsim/.

[13] R. L. Collins och L. Carloni, ”Flexible Filters: Load Balancing through Backpressure for Stream Programs,” i *Proceedings of the Seventh ACM International Conference on Embedded Software*, New York, NY, USA, ACM, 2009, pp. 205-214.

[14] M. Balazinska, H. Balakrishnan och M. Stonebraker, ”Contract-Based Load Management in Federated Distributed Systems,” i *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1*, San Francisco, CA, USENIX Association, 2004, pp. 15--15.

[15] V. Gulisano, R. Jiménez-Peris, M. Patiño-Martínez, C. Soriente and P. Valduriez, "StreamCloud: An Elastic and Scalable Data Streaming System," in *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, Piscataway, NJ, USA, IEEE Press, 2012, pp. 2351 - 2365.

[16] Y. a. Z. S. a. H. J.-H. Xing, ”Dynamic Load Distribution in the Borealis Stream Processor,” i *Proceedings of the 21st International Conference on Data Engineering*, Washington, DC, USA, IEEE Computer Society, 2005, pp. 791--802.