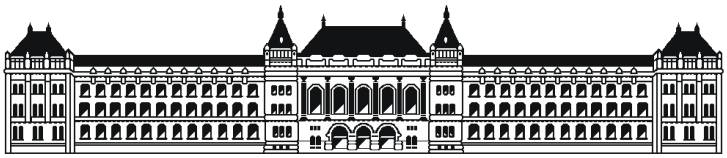

Budapest University of Technology and Economics
Department of Measurement and Information Systems

Distributed graph clustering engine

Master's Thesis

Author
Máté Szalay

Supervisor
András Kövi
research associate

May 10th, 2010

Abstract

The comparative analysis of complex interacting systems (such as biological, social or technical systems) with graph models becomes a popular research field in the last decade. One of the main challenges is to define the structure of the complex system by detecting the dense parts, so called communities or clusters of the large graphs. The huge number of different community definitions leads to dozens of graph clustering algorithms and lots of different implementations. The Parallel Graph Algorithm Framework (ParaGrAPH) was created to ensure the compatibility of community detection and other graph related methods. This framework gives a Java plug-in interface for the developers to implement distributed algorithms operating on large graphs. This thesis gives a short introduction of graph clustering definitions, shows the architectural concepts, implementation details and performance measurements of ParaGrAPH and describes the programming interfaces used for plug-in development.

Contents

Chapter 1

Introduction

The study of graph models (called networks) of complex natural or artificial systems has proved very successful to understand both their structure and dynamism. One of the most widely analyzed challenge in this field is the determination of community structure in the graph, i.e. the dense parts of the networks (called clusters, communities or modules), in which the network elements have a much stronger influence on each other than the rest of the network.

Network communities help the functional organization and evolution of complex networks. However, the development of a method, which is both fast and accurate, provides modular overlaps and partitions of a heterogeneous network, is rather difficult. I am working on the field of graph clustering together with some of my colleagues form the LinkGroup research group of the Semmelweis Medical University in Budapest, with the guidance of professor Péter Csermely. In one of our earlier publication [14] he collected the scientific publications related to graph clustering. The number of papers can be seen in Figure 1.1.. Many different concepts and cluster definitions exist, leading a huge number of different algorithms and methods [11, 8, 24].

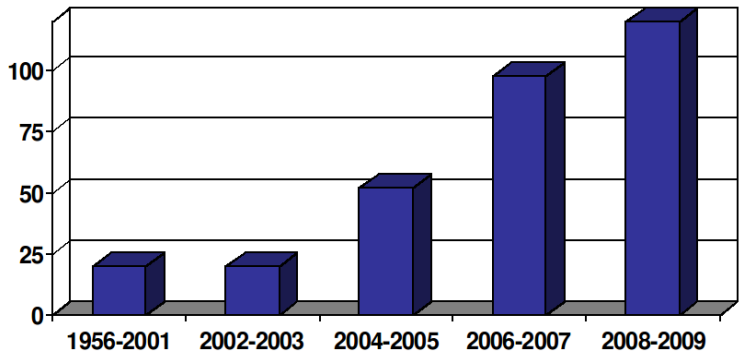

Figure 1.1: *Time-scale of the development of modularization methods.*

Parallel with this, some measurement and benchmark algorithm were developed by the scientific community to compare the different module structures determined by different methods [8, 16, 17]. This high diversity of methods solving the same base goal (i.e. graph community detection) makes very important to create a framework to help the scientists in this field to develop and compare their algorithms easier and more effective.

1.1 Parallel Graph Algorithm Framework (ParaGrAPH)

For the reasons described before, the Parallel Graph Algorithm Framework (named as ParaGrAPH from now) was created, capable to run user developed java plug-ins for clustering and benchmarking algorithms or performing other type of calculations on large graphs. The framework can be distributed to several computers, as well as used on one machine as a standalone program.

All the design and development works related to the ParaGrAPH framework were done by me during my last semester on the Measurement and Information Systems Department of Budapest University of Technology and Economics.

Chapter 2 will discuss shortly the existing definitions related to graph clustering and show the basic steps of a specific clustering algorithm family, called ModuLand.

The Chapter 3 gives a high level description of ParaGrAPH framework and shows the main architectural concepts by defining its building components and services.

Chapter 4 gives a closer look to some implementation details and design decisions, like the description of used open source tools and libraries, the introduction of plug-in and data distribution, life-cycle management, messaging techniques and the issue of availability in ParaGrAPH.

Then Chapter 5 will contain examples and a detailed how-to of plug-in development on ParaGrAPH framework. It will be shown how can the developer reach the distributed graph database or create periodic backups during the plug-in specific calculations.

Chapter 6 dealing with the question of performance. The results of measurements will be shown, executed to study how the run-time is affected by the changing of various parameters, like the size of the graph or the number of hosts used in ParaGrAPH.

Finally in Chapter 7, some of possible further improvements are gathered.

Some appendices are also attached to the thesis. In Appendix A.1 can be read the usage of running the ParaGrAPH system and GUI. While in Appenndix A.2 can be found the short description of the implemented members of ModuLand clustering method family used during the performance measurements.

Chapter 2

Graph clustering

In this chapter I want to give a short summary of the clustering and comparation methods developed recently. The definitions and the list of algorithms are based on the supplementary information of a paper [14] published by some of my colleagues and me. This paper defines a new algorithm family, called ModuLand, for detecting overlapping clusters in graphs. Some member algorithm of this family were implemented to the ParaGrAPH framework as well. More details about the ModuLand algorithms can be found for example in Section 2.4.

In one sentence the clusters can be defined as the dense groups of the network, in which the network elements have a much stronger influence on each other than the rest of the network. However, the correct mathematical or algorithmical definition of the clusters is much more complicated and diversified. In the next sections several definitions of graph clusters (called also as modules or communities) were gathered.

2.1 Local module definitions

Traditionally in the sociology and graph theory, several cluster definitions are based on 'local' topology. These definitions are usually quite exact in matematics point of view and can be usefull in some specific cases. However, in real life networks many times many of them have several weakness. According to the Supplementary Table S1 of [14] the following 'local' module definitions exist[1]:

- **Clique** is a complete subgraph of size k, where complete means that any two of the k elements are connected with each other.
- **K-clan** is a maximal connected subgraph having a subgraph-diameter less or equal to k, where the subgraph-diameter is the maximal number of links amongst the shortest paths inside the subgraph connecting any two elements of the subgraph [2, 23, 36].
- **K-club** is a connected subgraph, where the distance between elements of the subgraph is less or equal to k, and where no further elements can be added that have a distance less or equal to k from all the existing elements of the subgraph [2, 23, 36].

[1] some of the mathematical definitions were copied literally if the author was not able to find a more understandable one

- **K-clique** is a maximal connected subgraph having a diameter less or equal to k, where the diameter is the maximal number of links amongst the shortest paths (including those outside the subgraph), which connect any two elements of the subgraph [20, 2, 23, 36].
- **K-clique community** is an union of all cliques with k elements that can be reached from each other through a series of adjacent cliques with k elements, where two adjacent cliques with k elements share k-1 elements (please note that in this definition the term k-clique is also often used, which means a clique with k elements, and not the k-clique as defined in this set of definitions; the definition may be extended to include variable overlap between cliques) [3, 28].
- **K-component** is a maximal connected subgraph, where all possible partitions of the subgraph must cut at least k edges [22].
- **K-plex** is a maximal connected subgraph, where each of the n elements of the subgraph is linked to at least n-k other elements in the same subgraph [34, 36].
- **Strong LS-set** is a maximal connected subgraph, where each subset of elements of the subgraph (including the individual elements themselves) have more connections with other elements of the subgraph than with elements outside the subgraph [36].
- **LS-set** is a maximal connected subgraph, where each element of the subgraph has more connections with other elements of the subgraph than with elements outside of the subgraph [36, 7].
- **Lambda-set** is a maximal connected subgraph, where each element of the subgraph has a larger element-connectivity with other elements of the subgraph than with elements outside of the subgraph (where element-connectivity means the minimum number of elements that must be removed from the network in order to leave no path between the two elements) [6, 36].
- **Modified (weak) LS-set** is a maximal connected subgraph, where the sum of the inter-modular links of the subgraph is smaller than the sum of the intra-modular edges [36].
- **K-core** is a maximal connected subgraph, where the elements of the subgraph are connected to at least k other elements of the same subgraph; alternatively: the union of all k-shells with indices greater or equal k, where the k-shell is defined as the set of consecutively removed nodes and belonging links having a degree less or equal to k [33, 36, 13].

As we can see, these cluster definitions do not take into consideration the strength of a connection (weights of links in the graph), and many times there is a free parameter (called K for example) which defines the scale of the cluster size, or the sensitivity of the algorithm. It can lead to the well known problem, called giant component problem: these conventional methods often can not find big and small communities at the same time. Raise the detection limit too high, and find only the largest communities, or set the detection limit too low, where most of the overlapping, large communities are already merged. If the detection limit is small enough to find the smallest clusters, many times the whole network is collapsed to a single giant-component [5, 10, 15].

There are many ways to handle the giant-component problem. The effect of the continuous changing of the detection limit on the development of more and more details of the modular

network structure can be nicely followed in the hierarchical clustering methods (see later). Another solution to find both small and large communities is to simplify the network by leaving out the links below an appropriately selected arbitrary link weight threshold [28]. This network simplification makes the communities more isolated, and enables to lower the detection limit to see smaller communities but leaving larger communities still more-less separated showing only a reasonably minor overlap. A third effective way of dealing with the giant-component problem will be described in the ModuLand method family, see Section 2.4.

2.2 Global module definitions

Beside the local community definition methods, there are several clustering algorithms based on global measurements, such as the widely used modularity (Q) measurement. In these global definitions a benefit function is used to give the quality of a division of a network into modules or communities. For example good divisions has higher global measurements than wrong divisions. If there is an exact formula to compare two module structures, the whole clustering problem become an optimization question, and the challenge is to find the optimal clustering in the space of the possible module structures of the given network.

The original definition of the Q modularity value was given by Mark Newman [11, 27] *Modularity compares the number of links inside a cluster with the expected number of edges that one would find in the cluster if the network were a random network with the same number of nodes and where each link keeps its degree, but links are otherwise randomly attached.* More precisely:

$$Q = \frac{1}{2m} \sum_{ij} \left[A_{ij} - \frac{k_i k_j}{2m} \right] \delta(c_i, c_j),$$

where A is the adjacency matrix of the network (A_{ij} is defined as the number of links between node i and j), m is the number of the links in the graph and k_i is the degree of node i. The c_i is the group to which node i belongs and δ is the Kronecker delta symbol, so $\delta(c_i, c_j)$ is 1 if node i and j belongs to the same community, otherwise it is zero.

In a random network with same degree parameters, the expected number of links falling between two nodes i and j is $k_i k_j / 2m$, so the difference between the number of expected links and real links is $A_{ij} - k_i k_j / 2m$, summing over all pairs of vertices in the same group.

A huge number of cluster determination algorithm were developed to find the communities based on the original Q modularity measurement, or one of its variant. For example using hierarchical agglomeration (like split the original network into two module in a recursive way) with greedy optimization [25], or using random walks on graph [29]. Other works with multi-step greedy algorithms with vertex mover refinement [32], or using simulated annealing [12, 21], mean field annealing [18] or linear programming [1].

There are many different variation of the original formula above. These variations usually differs in the null-model, whose links are compared with the real network.

2.3 Other module definitions

In the last years a lots of clustering algorithms were developed based on neither local definitions nor optimization on global modularity functions. One new way is to define element similarity measurements and say the clusters are subgraphs containing element-pairs, which are similar to each other [19, 9, 8]. The similarity can based on distance, eigenvector components or even functional similarity coming from emergent network properties. However, this later often may add an element of redundancy to the definition, since the emergent function emerges many times from the modular structure itself.

An other interesting point of view is to define the cluster based on its information content, as a set of subgraphs allowing the greatest compression of network structure with a minimal loss of information [30, 31].

It is also possible to define modules based on the effect of simulated communication between elements of the network (for example information propagation, like gossip in the social networks, or perturbation flow in physical systems). In this case the communities can be defined as a set of elements, displaying a larger communication among them than to the rest of the network [28, 17, 35].

2.4 ModuLand clustering method family

Keeping in mind the huge diversity of cluster definitions, the members of the biological network related LinkGroup research group on the Semmelweis University developed the patented ModuLand method family, which gives common algorithmic structure for many of earlier module definitions, and gives the opportunity to even enhance them. The four basic steps of the ModuLand method family is shown on Figure 2.1.. This Figure and the description of main steps of the algorithm is based on [14], published by my colleagues and me.

Considering a real network as an interacting system, the quantitative simulation of the influence (or indirect impact) of a given node on the rest of the network is an interesting problem by itself. The first step of the ModuLand framework builds up these influence functions (we call them as *community heaps*) for each and every network element. A community heap can be extended over the whole network, which would produce accurate, but slow results, so practically it is beneficial to stop the influence simulation at a given threshold. The LinkLand algorithm is one way to generate these community heaps, the exact definition of this algorithm can be found in Appendix A.2.1. The ParaGrAPH plug-in version is used for different measurements in Chapter 6. As an example, Figure 2.1.A shows three community heaps defined over a co-authorship network published in [26].

As the second step of the ModuLand framework, the so called *community landscape* is generated by summing up all the community heap values of a given link of the network (which is the values of the influence functions of all other links of the network on the given link), and making this summation for each link. In order to give a visual representation, the resulting centrality-type values are plotted vertically over a 2D representation of the network resulting in a 3D visual image of the community landscape heights as shown on Figure 2.1.B . Now we can see hills and mountains of the community landscape consisting of those elements, which influence each other stronger than the rest of the network. This is exactly the same intuitive definition of modules, which was given in the second paragraph of this Chapter.

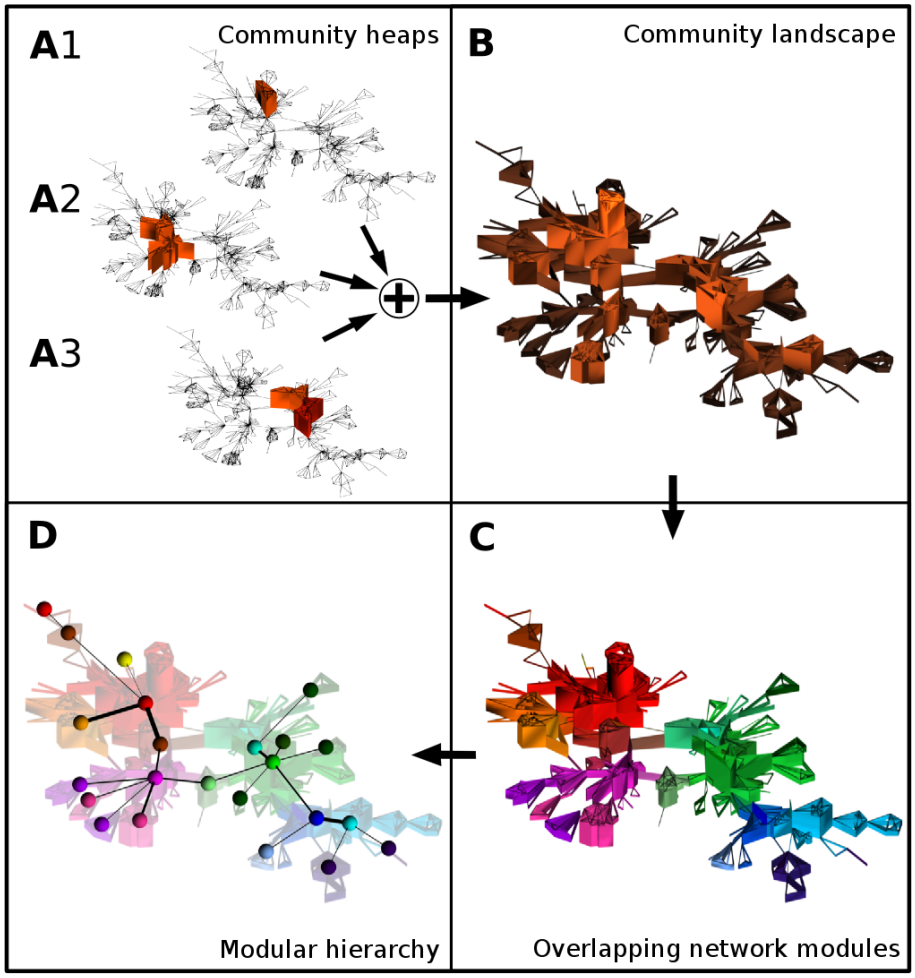

Figure 2.1: *The main steps of the ModuLand method family.*

The third and last step of the ModuLand method family identifies the modules of the network by finding the prementioned hills and mountains of the community landscape, and assigning the links and elements of the networks to these links (Figure 2.1.C). There are many different ways to construct an algorithm for identifying the hills of the community landscape, each of these algorithms is suitable for a range of applications, but not necessary for all of them. One of the developed methods is called as proportional module assignement method. The exact definition of this algorithm can be found in Appendix A.2.2. The distributed version of this algorithm was also created as a ParaGrAPH plug-in.

Optionally, a higher level hierarchical representation of the network can also be created, where the nodes of the higher level correspond to the modules of the original network, and the links of the higher level correspond to the overlaps between the respective modules, as can be seen in Figure 2.1.D. This hierarchical representation can be used recursively in several steps until the whole original network is represented by noninteracting elements, allowing a fast, zoom-in type analysis of large networks.

Chapter 3

Architecture

In this chapter I will describe the architecture of the ParaGrAPH framework and the function of its main components and packages. In the Figure 3.1. can be seen the high level view of the tree main components and their connections.[1] The framework is basically distributed into a server program and client programs. The server is mainly responsible for management and synchronization, while the clients running the distributed algorithms and calculations on the graph. The users can manage the framework trough a management GUI connected to the server. The ParaGrAPH system can be used also as a standalone program, when all the three type of programs are running on the same machine.

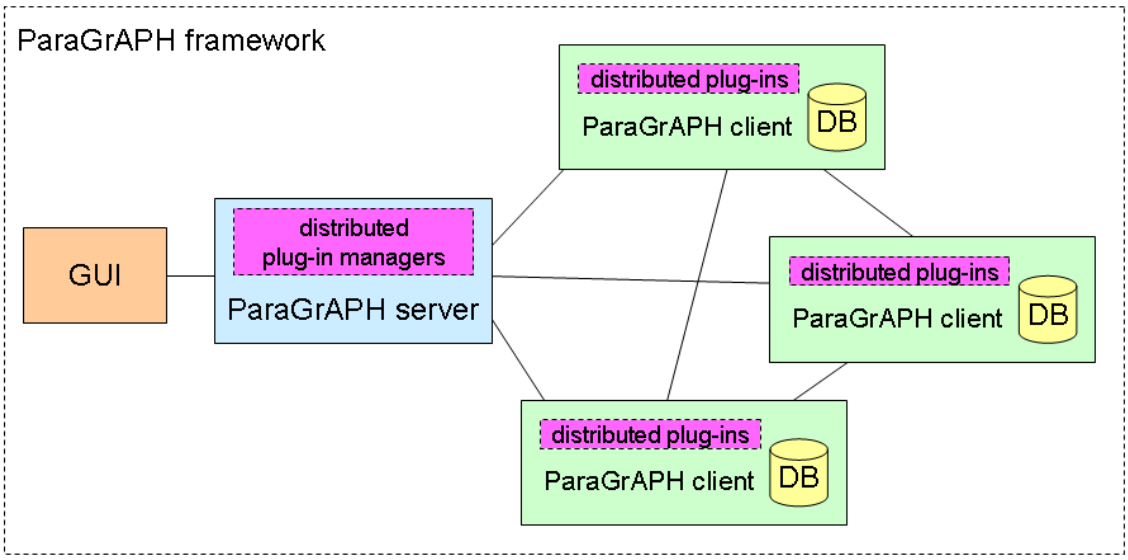

Figure 3.1: *High level view of ParaGrAPH system*

The aim of the system is to run distributed algorithms (such as clustering algorithms) on graphs. These algorithms are running on the system as user defined plug-ins. The developer of a new plug-in has to create a plug-in part which is running on the clients and makes the calculations, and an other part running on the server, responsible for managing the client plug-ins. All the client components have their own view of the current graph. The graph is

[1] The main components are marked with different colors in the same way in all figures of this thesis. The server releated parts are blue, the client parts are green and the GUI parts are marked with orange. The database related parts are colored to yellow, while the objects created by the plug-in developer are purple.

split into distinct parts, and each client is responsible for maintaining his part. Any given plug-in instance can reach directly the graph part which is maintained by its client, and with different operations he can see the whole graph, if it is needed. For more information on the distributed plug-in and graph database concepts, see Section 4.2.

3.1 Common layer structure

Both the server and the client component in the ParaGrAPH system follow the same layer structure, as can be seen in the Figures 3.2. and 3.3.. Three different layer are separated from each other. The lowest layer is called as control layer, the middle one is the Service layer and the third is called as the Plug-in layer. Each layer has common parts both in the server and client programs.

The Control layer has many responsibilities, such as:

- managing the life cycle of the program
- giving an entry point to the given program (the control layer is starting when the user starts the given server or client programs)
- initializing the other layers
- giving an universal messaging interface for the inter-component communication
- taking care of backup creation and restoration of the whole program
- giving a message-based synchronous event handler mechanism to the services

Most of these topics will be discussed later in this thesis. The Section 4.5 describes the backup creation, health check and availability issues, Section 4.4 shows the communication mechanism between components and Section 4.3 shows the state and sequence diagrams of the ParaGrAPH framework concerning also the question of service initialization and life-cycle management. However, the important common concept of message based synchronous event handler mechanism in the Control layer will be shortly discussed here.

In order to avoid synchronization problems and to make the state machine design of the services easier, the framework assures that only one service can be running in the given component at one time. A main event handler was introduced (as the technical realization, the Jade behavior scheduler was used, see Section 4.1). All events like parsing incoming and sending outgoing messages or changing states in case of any service or other component in the program, all these events are added to the main event handler queue of the Control layer. This makes the design easier, because only one thread is needed in case of the server, and only two in case of the client programs and the state transitions in the state machines of services can be executed as atomic operations.

The second layer is called as Service layer. It contains many components, each of them is offering a specific service to the other two layers. Some services are common both in the server and in the client programs, and some are different. The common services are:

- Logger service: Every component can use the logger service to log different type of events. Several type of events are distinguished, like *debug*, *normal*, *warning*, *status*,

statistics, *exception*, *alarm*, *database*, etc. There is a filter in the logger service which contains the currently valid event types. The filter can be altered in the GUI and the logs can be printed out to the console, or stored in separated files. The current structure can be easily enhanced by supporting database output or other distributed log storage.

- Context service: By this service, all the other components and services can reach each other inside a given server or client program. The different services (like the messaging and life-cycle management services of the control layer or the logger, plug-in runner and database handler services) are registered in the same singleton context object. It helps the components to find the currently valid service objects. For example there can be different realization of the logger module (like file logger, database logger, etc.). The active logger implementation can register itself in the local context service, and all local packages can retrieve this configured logger service from the context.
- Health Check service: Every program has the opportunity to check the health status of the others. Usually the client and GUI programs check the status of the server, while the server program checks the status of both of the GUI and the clients. For more information about health check, backup and aviability issues, please see Chapter 4.5.

There are some server or client specific services in the second layer, which will be described in Sections 3.2 and 3.3.

The third layer is called Plug-in layer. It contains the plug-in and plug-in manager objects representing the distributed graph algorithms running on the ParaGrAPH framework. These plug-in related objects are of course different on server and on client programs. However, the package called plug-in factory is the same. It is responsible for creating the plug-in and plug-in manager objects based on the same identifier, contains the name of the plug-in and the name of the plug-in type (so called *package prefix*). The plug-in objects contains the distributed algorithm itself. These objects are initialized and running by the plug-in runner component on client side. The plug-in manager objects on the server side are responsible for the synchronization and management of these plug-in objects. The plug-in manager objects are handled by the plug-in scheduler service in the ParaGrAPH server program. For more information about the plug-in handling mechanism see the Section 4.2 and for the ways of plug-in development see Chapter 5.

3.2 ParaGrAPH server

The architectural view of ParaGrAPH server program can be seen in Figure 3.2.. This program is responsible for synchronizing the clients' life-cycle. For example in the beginning of a new project the server program registers the available clients for work, broadcasting them the unique client IDs, sending them the requests for loading their last backup, running plug-ins, making them quit, etc. (more details about the life-cycle management can be found in Section 4.3).

The user management component communicates with the GUI. In the current implementation only one GUI can be connected at one time. The server is acting as a bridge to send the alarms and status messages of the clients and plug-ins to the management GUI. The GUI can disconnect from and reconnect to the server at any time, it will not effect the current work of the system.

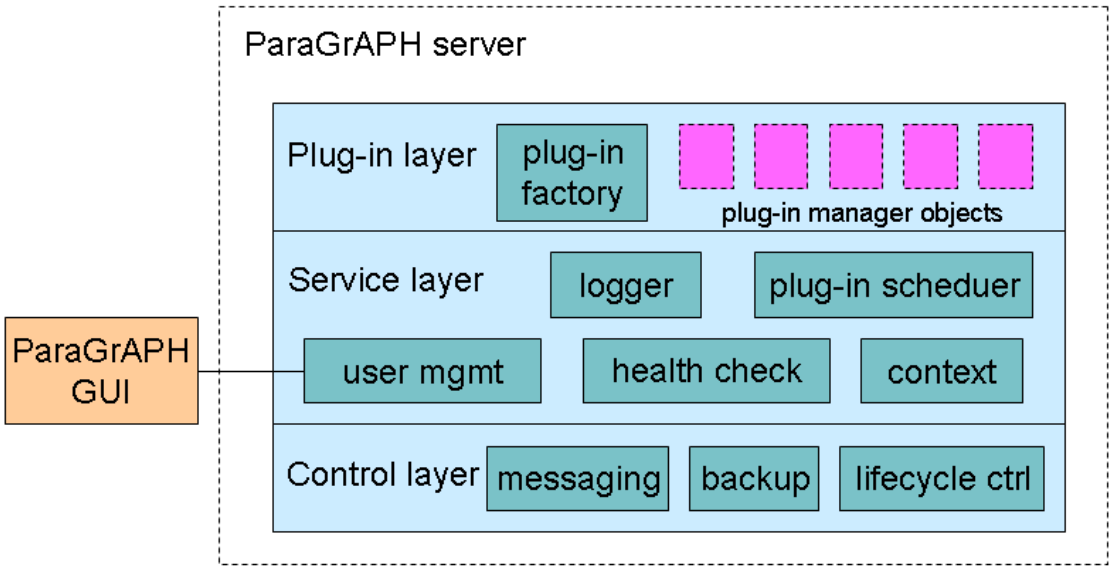

Figure 3.2: *The architecture of the ParaGrAPH server component.*

The system can work only on one graph structure at a time. However, one graph can contain more than one separated subgraphs (components), so it up to the plug-in developer to make calculations on different networks. But there is no possibility to run two different plug-ins in one ParaGrAPH system simultaneously. The plug-in scheduler component in the server program maintains an algorithm queue to store the requests coming from the GUI. This package coordinates the state machine of the client plug-ins, and indirectly the state of the distributed graph database too.

The graph data from the distributed graph database can not be reached by the server program, but all the client plug-in can send back result information during its work, which can help the plug-in manager to manage the work and make decisions. The plug-in manager objects created by the developer using ParaGrAPH framework contains the business logic of plug-in management. Every plug-in can contain several distributed jobs. The plug-in manager object decides which clients run which jobs, they can set the parameters of the jobs separately for each client and retrieve the result information after the running of each job. More information about the plug-in and data distribution can be found in Section 4.2, and about the plug-in development relating interfaces and working code examples in Chapter 5.

The whole ParaGrAPH system creates backup files periodically. In case of a failure the whole system can be restarted from the last saved state. Every backup on server side contains the state of the plug-in manager object, the unique ID of the current project and the number of connected clients. During the loading of backups, the server waits until every client restored its own last state of the given project before it starts the plug-ins. The backups are saved in two phase to ensure there is always one valid backup file even if the error happens during the creation of the backup. For more information about the availability issues please see Section 4.5.

3.3 ParaGrAPH client

The main components of the ParaGrAPH client can be seen in Figure 3.3.. Beside the common components described in Section 3.1 there are also several client specific functions. After starting a client, it will register itself in a global yellow-pages service and waits for the server to select him for work (see Section 4.3). Every client has an unique ID number to distinguish the different clients in the plug-in manager objet on server side. This number is used also in clients to identify the owner of the distributed database objects and also appears in the logs to make the debug activities easier. The unique ID is saved in the backups of the clients too, so it will remain the same during the whole project. The unique ID for the given client is decided and sent by the server.

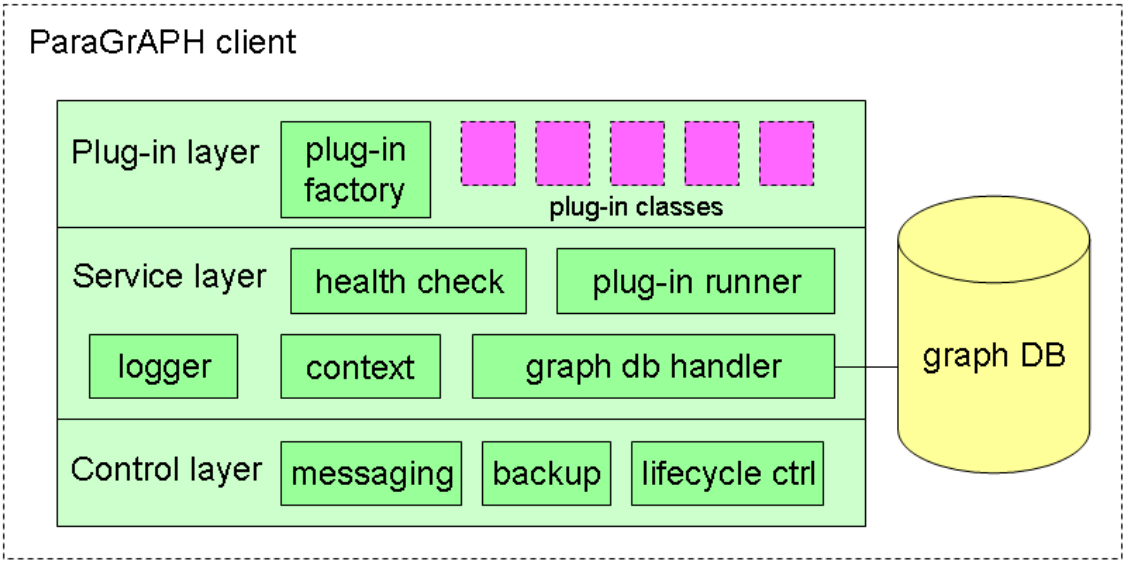

Figure 3.3: *The architecture of the ParaGrAPH client component.*

The client programs are responsible for running the distributed algorithms developed as plug-ins in the ParaGrAPH framework on the distributed graph database. The graph related data can be reached in the plug-ins by using the graph database handler service. In the current implementation no real database is used, but the graph related data is stored in an optimized object structure in the memory. Every object in the graph database has an owner client. If the plug-in reads or changes a local database object, the database handler can simply retrieve this object from the local store. If the plug-in works on a non-local data object, then a cached copy is maintained by the database handler locally. Every plug-in may contain several jobs, and every change on the local cached data object copies will be committed to their real owner simultaneously by all the clients in the end of every distributed job. This behavior of the distributed database basically affects the state machines of the database handler and plug-in runner components, and very important to understand for the plug-in developers in order to create optimized distributed algorithms. In the whole ParaGrAPH system only one specific job of a given plug-in can run at a time. The synchrony is maintained by the plug-in manager service of the ParaGrAPH server, which requests to loading of the given plug-in on the client and the running of the given job inside this plug-in. The commit phase starts after all the requested clients finished their job. More information about the plug-in and graph database distribution can be read in Section 4.2, and about optimized plug-in development in Chapter 5.

The clients also create backup files separately. The ParaGrAPH framework creates a backup

automatically in the beginning of every job. This backup contains the project, the plug-in and the job identifiers, the current view of the distributed graph database and the parameters of the job sent by the plug-in manager object from the ParaGrAPH server. The developer of the plug-in can also asks the framework to create a backup file during his algorithm (it is highly recommended if the job requires long time to run and its state can be saved and restored relatively easily). However, in this case the plug-in developer needs to define the state saving and loading procedures of his plug-in. The description of this backup handling concept can be seen in Section 4.5, and a working example is shown in Section 5.3.

3.4 ParaGrAPH GUI

The ParaGrAPH GUI program is basically a simple user interface, responsible for:

- locate and connect to the server program
- send the name (identifier) of required plug-ins and their parameters to the server
- retrieve from the server and show to the user the status messages and alarms

The ParaGrAPH GUI program do not share the same layer structure as the server and client programs. It contains two separated layers, as can be seen in Figure 3.4.. The first (bottom) one is communicating with the server program while the second is for realizing the graphical functions and dialogs. The communication with the server is happening trough a fixed message protocol, called management protocol. So in the future just by changing the second layer it will be possible to create other management interfaces for command line use, or for processing project descriptor XML or text files.

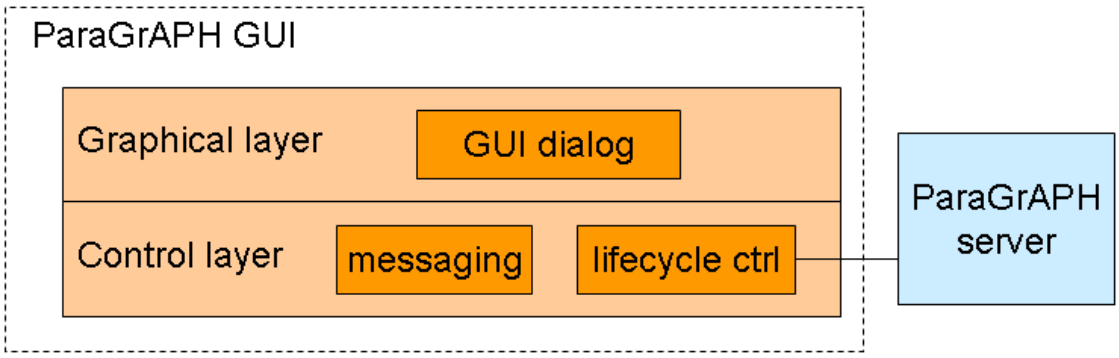

Figure 3.4: *The architecture of the ParaGrAPH GUI component.*

Currently the plug-ins can show their results by changing the distributed graph database, creating data files, logs or status messages toward the GUI. However, in the future it would be a nice improvement if the plug-ins could embed their own graphical interface to the ParaGrAPH GUI component.

The usage of ParaGrAPH GUI with screenshots can be found in Appendix A.1.

Chapter 4

Implementation

The whole ParaGrAPH framework is implemented in Java, makes it fully platform independent. No commercial program was used during the development, just the latest versions of Java, Ant, Eclipse IDE and JADE, could be reached in the beginning of development. More precisely:

- Sun Java 6 (build 1.6.0_18)
- Apache Ant 1.8.0 (build on February 1 2010)
- Eclipse Galileo (Build id: 20100218-1602)
- JADE 3.7 (01/07/2009)[1]

In the near future the project will be uploaded to the SourceForge[2] open source software directory under LGPL[3] license. In Appendix A.1 can be found more information about the different Ant targets responsible for building the jar file and starting the ParaGrAPH framework.

4.1 Java Agent Developement Framework

This thesis does not contain the accurate description of the **J**ava **A**gent **De**velopment Framework, called JADE. The following brief introduction is based on the technical descriptions, programming tutorials and other documentations can be found on the homepage of the project (`http://jade.cselt.it`) and contain much deeper specification of JADE.

The JADE framework is a distributed java platform to develop agent based programs compliance with the FIPA standards. FIPA by its own definition[4] is *the Foundation for Intelligent Physical Agents, an IEEE Computer Society standards organization that promotes agent-based technology and the interoperability of its standards with other technologies.*

The JADE runs agent objects, may be distributed on several host, and contains many services to support the functioning of these agents, like: naming service, yellow-pages service, inter-agent communication service and predefined interaction protocols. On each host

[1] the latest JADE version is 4.0 now (last build: 20/04/2010)
[2] `http://sourceforge.net`
[3] GNU Lesser General Public License: `http://en.wikipedia.org/wiki/LGPL`
[4] `http://www.fipa.org`

only one JVM[5] needs to run, contains the JADE framework together with agents. In ParaGrAPH, three agents were developed: the server, client and GUI agents.

Each agent in JADE has its own Java thread and can contain several tasks, called behaviors in JADE term. These behaviors can be run in different threads or can be running sequentially secluded by the agent's own cooperative mechanism, where by default the tasks are not interrupted by the agent. ParaGrAPH usually uses the JADE scheduler to ensure the safe and effective operation. However on client side a separate thread is needed for running the plug-in objects and handling the synchronous update of distributed database objects (see Figure 4.3.).

4.2 Plug-in and data distribution

The concept of algorithm distribution in ParaGrAPH (see in Figure 4.1.) is quite similar to the Google's map-reduce technique. The 'map' part is done by the clients in the plug-in objects, while the 'reduce' parts by the server in the plug-in manager objects (and by the distributed graph database). According to this concept, the original problem is solved by splitting the algorithm into steps, so called jobs in ParaGrAPH. The clients working on a job together then send their results back to the server. After every client is done, the server checks the results, make its decisions and request the clients to run a new job.

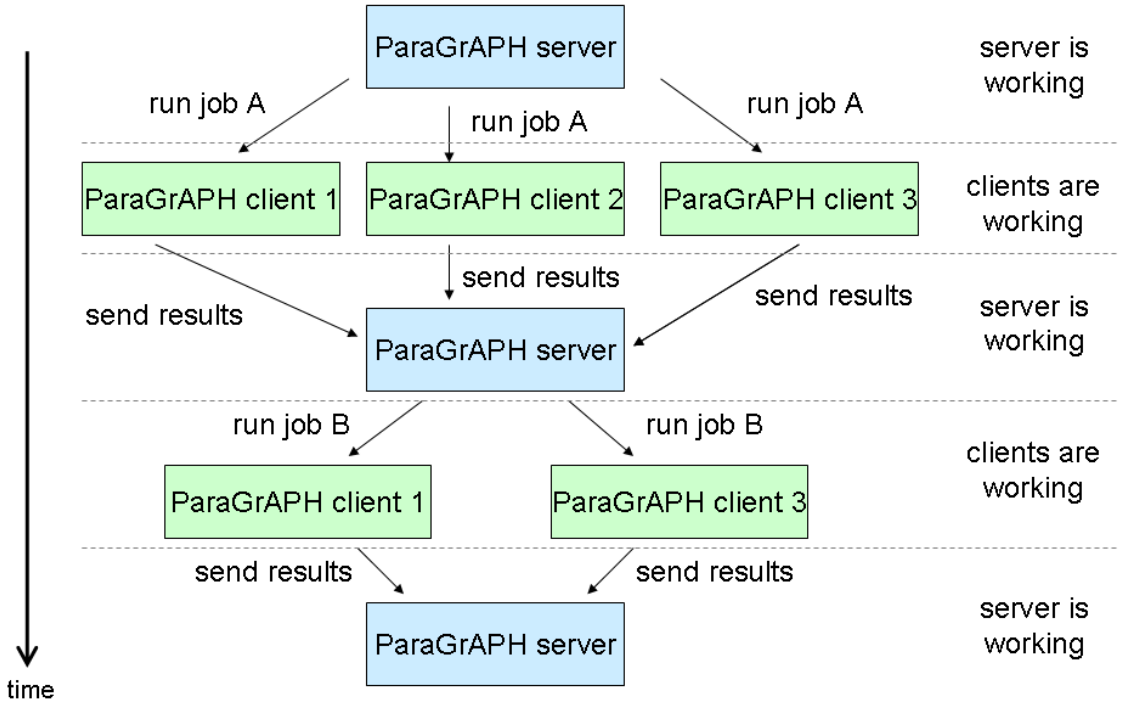

Figure 4.1: *The algorithm distribution concept in ParaGrAPH.*

During the run of a job, all the client has his own view of the graph database. The modifications done by the given client are hidden from the others. After all clients finished the current job, in the 'reduce' phase the modifications made by the clients are all merged to the distributed database. The simplified version of the state machine used in ParaGrAPH plug-in scheduler can be seen in Figure 4.2..

[5] Java Virtual Machine

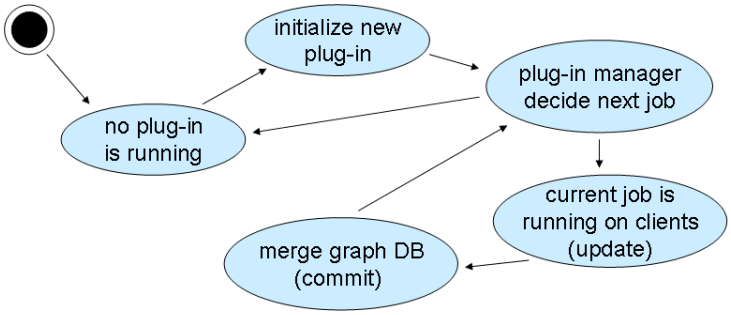

Figure 4.2: *Main states in the plug-in scheduler component of the server program.*

Initially no plug-in is running in the system. The plug-in scheduler component in the server program maintains a queue to store the plug-ins requested by the user. Only one plug-in can be active at one time in ParaGrAPH.

When a new plug-in is about to start, the scheduler has to initialize it on the server and asks the clients to do the same. The state can not be changed until both the server and the clients finished their initialization. Among others, here happen the creation of plug-in manager object (on server side) and plug-in objects (on client side) by the plug-in factory. These objects were implemented by the developer of the current distributed algorithm. The factory requires the name of the plug-in and the package prefix in order to create the objects both in client and server side.

After everyone finished the initialization, the plug-in object on the server decides which job will run. Only one job can be active at one time, but it is not necessary for all the clients to be active and run the selected job simultaneously. For example in case of I/O plug-ins, it is possible that one client is enough to run the given job. The plug-in manager is responsible for selecting the next job and for the decision of which client will be active. Based on the plug-in manager's commands, the plug-in scheduler service sends the run job requests to the selected clients. Each client can get also different parameters among the ID of the active job.

While the clients are working on the current job, the server is waiting. When the plug-in is running on a selected client, it operates on the distributed graph database. All the objects stored in the DB belongs to a client which is responsible for it. In the given client's point of view an object is called local, if the same client is responsible for it. When the plug-in reads or writes non-local objects, it is possible the objects are not up to date. It needs to be asked from the responsible client. For example in Figure 4.3. the plug-in running in client A wants to read a non-local edge object E. First the database handler recognizes it has no up-to-date version of the requested object. Then it stops the job thread in client A and send an update request to client B who is the owner of object E. Client B respond the request of client A and sends back the valid version of edge object E. This version shows the state of edge object E at the very beginning of the job, so it does not contain the possible modifications on this object during the current job. When the database handler in the main thread gets the response, it updates his edge object E in his local storage and wakes up the job thread in client A. The job can now continue its work.

If the jobs are operating on a large amount of non-local object, this update mechanism can take a lot of time. In this case it worth to update the whole network by some large query and not to ask it in case of every single object. The later technique is also supported

by the ParaGrAPH system. The desired behavior can be selected in the GUI among the parameters of the plug-ins. The two update mechanisms are compared in Section 6.4.

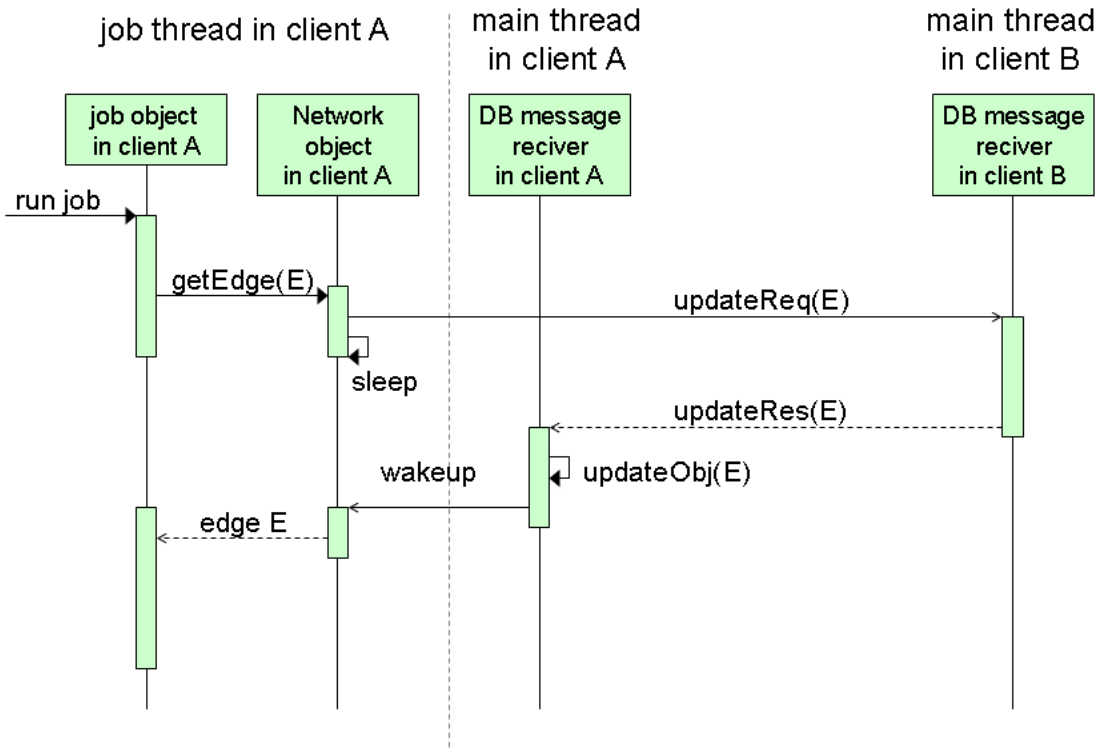

Figure 4.3: *Update mechanism on distributed database object during the run of jobs on clients.*

After all requested client sent back the results of the last run to the server, the plug-in scheduler directs the clients to commit their changes on non-local objects to their owners. If more than one client changed the given object during the job, it is nondeterministic which client's view on the object will survive, but one of them will completely. One exception is allowed: if the plug-in developer marks a property as 'cumulated', then all the changes will increase the original value of the given property. For an example please see the 'height' values in implementation of LinkLand plug-in. More information about the available operations on the distributed database can be read in Section 5.2.

If every clients finished the database merge phase, the plug-in manager object on the server side decides the next move, based on its own business logic and the results of the jobs sent back from the clients. It can run an other job, or finish the plug-in too.

4.3 Life-cycle management

Both the server and client programs use the same main concurrency model, inherited from the JADE Agent class. The concurrent events are added to the event handler queue and handled sequentially. The scheduling of behaviors in JADE is not pre-emptive (like in case of Java threads), but cooperative. It means when a behavior is scheduled for execution, there is no guaranty it will be ever finished to let the other behaviors work. This is one of the reasons why the client has two separated Java thread, one for the ParaGrAPH framework itself and one other is given to the developer of the distributed plug-in.

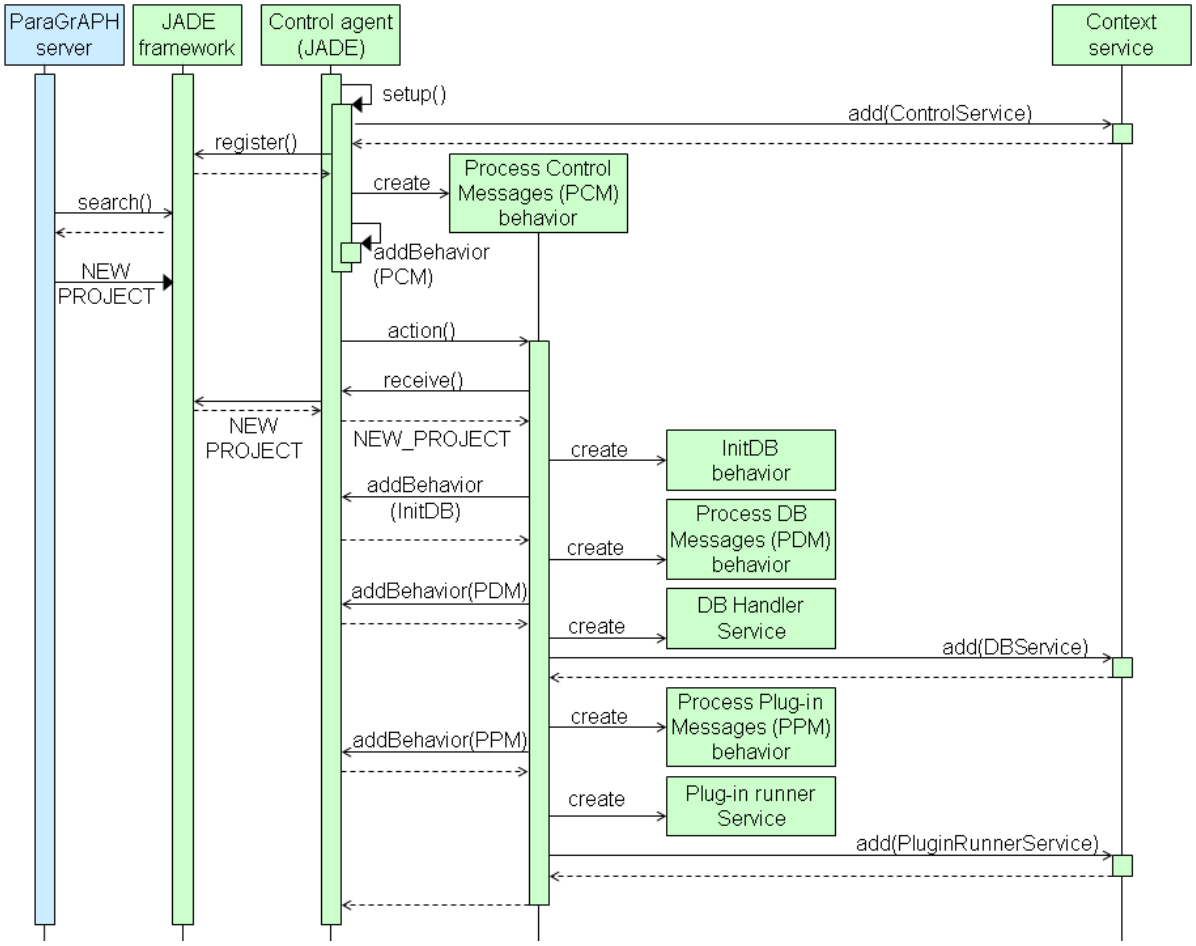

Figure 4.4: *The sequence diagram of a ParaGrAPH client start up.*

In Figure 4.4. can be seen the initialization of the most important behaviors in the ParaGrAPH client program. When the user starts the client with the given Ant target, the JADE framework will be initialized and the control agent of the client will be started. First the JADE will call the overridden *setup* function of the agent, which will register the client in the yellow-pages service of the JADE, and creates a new behavior named $ProcessControlMessages$, surprisingly having the responsibility for processing the control messages came from the ParaGrAPH server. The *setup* function also register the control service (what is basically the control agent itself hided with an interface) in the Context service, which is a local static service reference storage for the given client.

When the server wants to create a new project, first makes a lookup in the JADE yellow-pages to find the available clients. The server has a built-in limit set to 20 (which should be a changeable parameter in the future) on the maximal number of clients in a given project. The server chooses the first 20 available clients and sends them a NEW_PROJECT control message. The $ProcessControlMessages$ object is a cyclic behavior, only activated when a control message arrives. If there is no control message in the message queue, the behavior goes to the *waiting* JADE state. When the server sends the NEW_PROJECT control message, the $ProcessControlMessages$ behavior in the client will wake up and receive the message from the JADE framework.

Then it will initialize a new project by closing the old services (not shown in the figure) and create new database and plug-in service related objects. It creates two separate behavior for processing the database messages from other clients and plug-in messages from the

plug-in scheduler service on the server. These behaviors will be added to the control client agent's JADE scheduler. New objects for the second layer services are also created here, like in case of the Database Handler and Plug-in Runner services, or the Logger service not shown here. And before everything else, a new one time running behavior is created and added to the agent, which behavior is responsible for making some initialization in the distributed graph database right after all the services are properly created. Every service registered itself in the Context object to make itself reachable from any other services on the given client.

4.4 Messaging between components

The messaging between ParaGrAPH server, client and GUI programs is done over the inter-agent communication service of the JADE framework. Among others, it supports the asynchronous sending of $ACLMessage$ objects to one or more recipients. In our project this message object is logically covered with a general abstract message object called $ParagraphMessage$, which contains several information (like message type, string map with parameter name and value pairs and possible binary content as serializable object), and can be used as base class for the extended service specific message objects.

Five kind of logical message channels are used in the ParaGrAPH framework. Each channel is marked with a so called *ontology identifier* in the JADE ACL messages to help the separation of incoming message events by services. Aligned with the concurrency model used in ParaGrAPH, all the sending and receiving of messages are basically different events, named behaviors in JADE term. These behaviors are added to the queue of the scheduler used by JADE for each agent. For all service wanted to communicate with other components in ParaGrAPH, a cyclic behavior is defined for processing its own messages.

Figure 4.5. shows the main concept of messaging and the message channels in case of the server-client communication. The JADE and the messaging service in ParaGrAPH together taking care of the logical channels look to be separated for the services communicate trough them. The five logical channels are used for transmitting the following type of service specific messages:

- Plug-in messages travel between the Plug-in Scheduler on server and Plug-in Runner on client side. Basically the Scheduler sends here requests to the client for starting the different phases of plug-in running (initialize plug-in, running job, doing database merge, etc.). Usually the server needs to wait for the responses of all client before sending the next request.
- Control messages are used for the life-cycle management of the ParaGrAPH framework (creating new project, restoring backup, closing the framework, etc.) This channel is also running between the server and every client.
- Health-check messages are used for monitoring the status of server, client or GUI programs by the others.
- Database Messages travel only between clients to update or commit distributed database objects. The asynchronous commit messages contain a set of distributed database objects, currently maximum 200 objects in one message. (This parameter should be free to change in the future) The update mechanism happens during the run of jobs and needs to be synchronized. There are two different update request

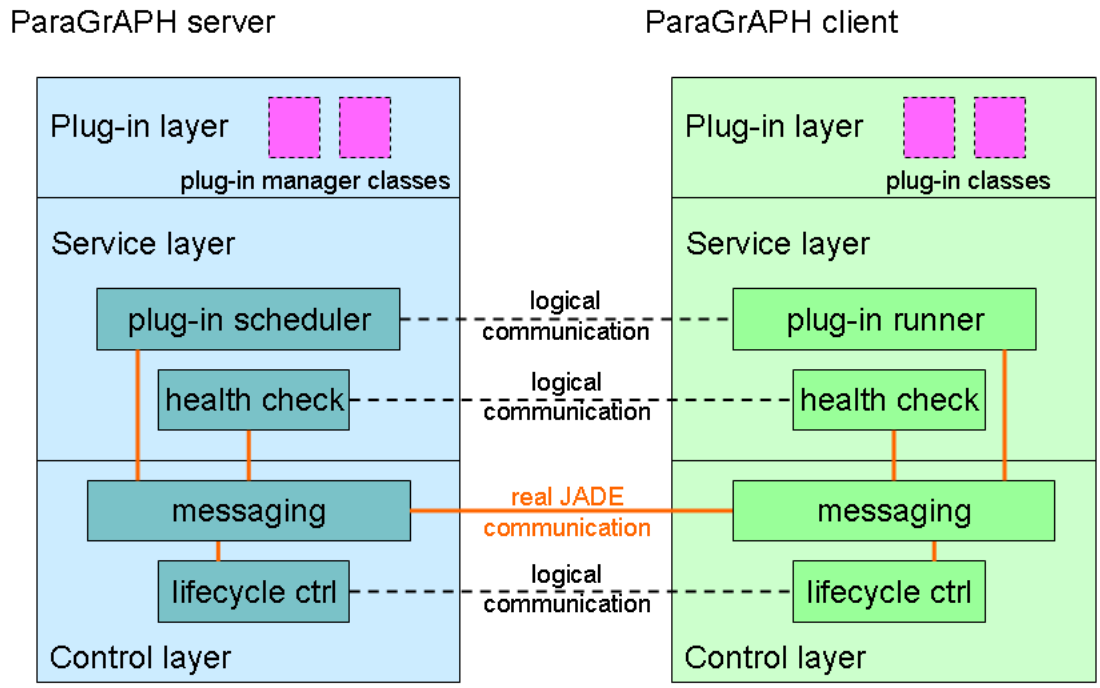

Figure 4.5: *The communication between the ParaGrAPH server and client programs.*

messages. One requires the update of a specific object sending its identifier and current local version number. In case of the other mechanism all the database object owned by the given recipient is requested with one signal. The response in this case is also a set of object.

- Management Messages are used in the communication between the server and the GUI. The GUI can connect and disconnect from the server, start new projects, restore backup, run different plug-ins or close the whole ParaGrAPH framework with the help of the management messaging protocol.

4.5 Availability

On availability point of view, the distributed ParaGrAPH system architecture has lots of challenges to be solved. Some hints on possible single point of failures:

- There is only one server in the system responsible for the whole synchronization (no hot swap at the moment).
- The messaging service used in ParaGrAPH relies entirely on JADE. If a lower level JADE message lost, there is no function of resending in ParaGrAPH and one missing message in most cases leads to the blocking of the whole system.
- The database objects can be cached in many clients, but only one has the up to date primary replica. If a client fails, all of the distributed graph database object he is responsible for will be lost.

- The ParaGrAPH clients and server run plug-in code not verified during the development of the framework. There is no automatic process now to handle if a plug-in manager object on server side or a plug-in object on client side fails for example with an exception or going to an infinite loop.
- etc.

For ensure the high availability and fault tolerance, these issues should be considered among others. Some of them can be soften or solved by relying on JADE solutions[6] released in the earlier or recent version 4.0. (In the beginning of the development only version 3.7 was available.) While the solving of other issues will be one of the future enhancements in the next ParaGrAPH versions.

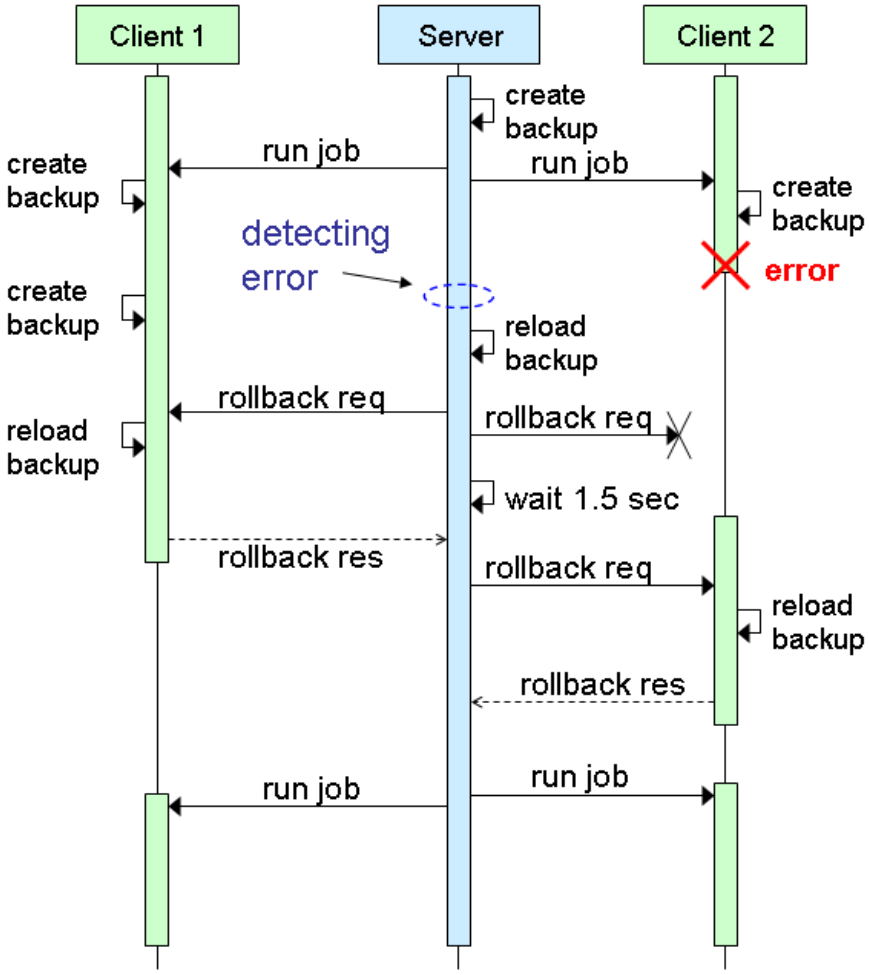

Figure 4.6: *Restoring backup after error during long time plug-in calculation.*

The current version of ParaGrAPH is not for supporting mission critical graph clusterization with "five nine" availability. However, the most of failures can possibly cause the loss of an unacceptable amount of time in case of huge calculations take many days or even more. For minimizing these time loss we introduced an automatic backup creation mechanism and a currently manual restoring function. The basic idea is to save the state

[6] like the enhanced *MainReplicationService* with support for application specific checks on connectivity status when a the primary JADE framework (main container) replica is considered dead due to a long disconnection

of the system in the beginning of each distributed job and give the possibility to the plug-in developers to create personalized checkpoints into their plug-ins (see Section 5.3). The sequence diagram of Figure 4.6. shows a possible failure scenario.

The backups are created to the file system (preferably to some reliable local or network storage). On server side it contains the identifiers of the current project, plug-in and job, together with other state variables (like for example the serializable result objects optionally sent by the clients in the end of the last job, etc.). The ParaGrAPH framework automatically creates backups on each client right before the start of any job. Among others it contains the whole current view of the distributed database, the project, plug-in and job identifiers and the incoming job parameters sent by the server. If an error happens, the system can be restored to continue the calculations on last saved state. In worst case on the beginning of the last job. However, in case of long jobs it is highly recommended for the plug-in developer to initiate backup creation during the calculations.

There is a health check service implemented in ParaGrAPH to give the possibility for each program (like the server, clients and GUI) to monitor the status of the others. This function is basically for detecting a failure of one or some program (a client or server host). In case of a client error, the server can start the restoring procedure automatically, while in case of a server error the client components can make a last backup and then wait for the beginning of a restoring procedure by the new server or make a graceful shutdown. However, this health check function is disabled by now, because with the current JADE agent implementations many times in case of the failure of one JADE host, the whole JADE system collapses with all ParaGrAPH server, client and GUI programs, making the status monitoring functionality quite unnecessary. There are several way to solve this problem in the future versions of ParaGrAPH, like using enhanced JADE techniques or using non-JADE health check and availability management services (e.g. SAForum[7] implementations) to monitor the state of ParaGrAPH hosts and restarting them in case of a failure.

[7] Service Availability Forum: `http://www.saforum.org`

Chapter 5

Developing ParaGrAPH plug-ins

This chapter contains a simplified tutorial on developing plug-ins to the ParaGrAPH framework. For creating a distributed algorithm, there is no need to see trough the implementation of the whole ParaGrAPH system. However, for the optimal performance it may help to understand the main concepts, especial the issues of plug-in and data distribution (see Section 4.2).

The development of a ParaGrAPH plug-in has three practical steps:

1. implementing a distributed plug-in object (will be run on client programs)
2. implementing a plug-in manager object (will be run on server program)
3. make sure the same version of the new objects are distributed on all ParaGrAPH instance

Currently there is no version control of plug-ins, and no central spreading mechanism to propagate (or even test) the developed plug-in class files trough the system, so at the moment it must be done manually.

5.1 Simple example: degree distribution

The basic concepts of plug-in development will be introduced on a simple 'hello world' like algorithm. Here can be find the java source of a plug-in to calculate the degree distribution of a graph (where degree of a node means the number of the links connected to it):

```
1 package hu.linkgroup.paragraph.plugin.general;
2
3 import java.util.HashMap;
4 import hu.linkgroup.paragraph.framework.services.logger.Logger.LogType;
5 import hu.linkgroup.paragraph.moduledb.interfaces.Node;
6 import hu.linkgroup.paragraph.plugin.DistributedJobBase;
7 import hu.linkgroup.paragraph.plugin.DistributedPluginBase;
```

The class is located in the `hu.linkgroup.paragraph.plugin.general` package. Every plug-in class must be somewhere in the `hu.linkgroup.paragraph.plugin` package, but the developer can create any sub package here. Both the plug-in and plug-in manager objects must be in the same package.

```
9  public class DegreeDistribution extends DistributedPluginBase {
10
11   // the static job IDs:
12   public static final int CALC_LOCAL_DEGREE_DIST = 0;
13
14   public DegreeDistribution() {
15     super();
16
17     //add all the job  to the plug-in:
18     addJob(new CalcLocalDegreeDist(),CALC_LOCAL_DEGREE_DIST);
19   }
```

Every plug-in class must extend the `DistributedPluginBase` class, and overload its constructor, as can be seen in the example. The degree distribution calculator plug-in has only one job, but usually more than one job is needed to cover a complex distributed algorithm. Every job has a static positive integer ID, used for example by the plug-in manager to request the clients to run the job with the given ID. The jobs are represented as objects too, and there is a map maintained by the `DistributedPluginBase` object, where the job ID and job object pairs are stored. The job object instances are added to this map in the overloaded constructor. The plug-in class must not contain any information about the state of the current plug-in, because only the state of the active job will be saving and restoring during the backup handling (see Section 5.3).

It is a good practice to implement the distributed jobs as inner classes of the plug-in object, if the length of these objects does not suggest to separate them into different java class files. The `CalcLocalDegreeDist` class is defined here:

```
21   public class CalcLocalDegreeDist extends DistributedJobBase {
22
23     @Override
24     public void runJob() {
25
26       //print out some debug log...
27       logger.log("starting job:" + getJobName(), LogType.DEBUG, this);
```

Every job class must extend the `DistributedJobBase` class, and overload at least one of its function, called `runJob`. This function will be run by the plug-in runner component of the client program and it is responsible for running the distributed algorithm itself.

The distributed job object can reach the logger service of the ParaGrAPH client. In line 27 the `logger` service is used to create a debug type log event. The `getJobName()` method gives back the name of the current job. It is used by the framework only for logging purposes, it can be overloaded to give some more sophisticated information about the job. By default it gives back the simple name of the job object.

```
29       // create the map contains the result distribution
30       HashMap<Integer,Integer> degDist = new HashMap<Integer,Integer>();
31
32       //go trough the local nodes
33       for(Node node : network.getLocalNodes()) {
34         int degree = node.getConnectedEdges().size();  //get degree
35
36         //get the current frequency of the given degree
37         int frequency = 1;
38         if(degDist.containsKey(degree))
39           frequency += degDist.get(degree);
40
```

```
41            degDist.put(degree, frequency);  //save new frequency value
42          }
```

When the plug-in manager sends the job run request to any client, it can send a parameter list too. This parameter list is stored in the parent class of the distributed job instance in the variable named `parameters`. The type of this object is `Map<String,String>`, and containing the parameter name and value pairs sent by the server when requires to start the given job. It is not necessary for the server to specify parameters, in which case the value of the `parameters` variable is `null`.

The local view of distributed graph database can be reached trough the `network` object inside the job class (see line 27). There are numerous handful functions to get the information about the node, link and module objects. For example the `getLocalNodes()` function gives back a set of references to the local node objects. The detailed description of the database interface can be read in Section 5.2.

```
43          jobFinished(degDist); //set result
44        }
45      }
46    }
```

After the plug-in finished the counting of node degrees in the last code line of the `runJob()` function, it sets the result object to send back to the server (see line 43), and informs it about finishing the job. The `jobFinished` function has one parameter, the serializable result object. If this function gets `null` as parameter, it will inform the server there is no result object in the end of the given distributed job instance. It is possible that some clients send back result and some others not in case of the same job.

The plug-in objects are running on client side, but every plug-in is synchronized by a plug-in manager object on server side (see Section 4.2). The source code of the later is starting below.

```
1  package hu.linkgroup.paragraph.plugin.general;
2
3  import java.util.HashMap;
4  import java.util.TreeMap;
5  import java.util.Map.Entry;
6  import hu.linkgroup.paragraph.framework.services.logger.Logger.LogType;
7  import hu.linkgroup.paragraph.plugin.DistributedPluginManagerBase;
```

The `DegreeDistributionManager` object has to be located in the same package as the plug-in object, always somewhere inside the `hu.linkgroup.paragraph.plugin` package. In the GUI, when the user wants to run this plug-in, he will need to give two things as parameter: the name of the plug-in ('*DegreeDistribution*' in this case), and the package prefix (what is '*general*' now, according to the line 1).

```
8
9  public class DegreeDistributionManager
10                    extends DistributedPluginManagerBase {
11
12   @Override
13   public void runNextJob() {
14     super.runNextJob();
15
16     // a simple state machine, where the
17     // state is coded with the actual job ID
18     switch(actJobId) {
```

```
19      case NO_JOB: //no job run yet
20        actJobId = DegreeDistribution.CALC_LOCAL_DEGREE_DIST;
21        for(int i=0; i<clientNum; i++) {
22          pluginScheduler.sendRunJobReq(i, actJobId,
23                      /*jobParameters*/ null, clientNum);
24        }
25        logger.log("local degree distribution jobs started",
26                                LogType.DEBUG, this);
27        break;
```

The plug-in manager objects must extend the `DistributedPluginManagerBase` object and override at least one function, called `runNextJob`. This function contains the business logic of the plug-in manager. It will be called each time, when the plug-in scheduler on the server needs to make a decision about what to do next (see Figure 4.2.). It is called in the beginning of the plug-in after the initialization, and every time when the clients finished the current job and the merge phase is over. Basically it is a simple state machine, where the state is stored in the integer variable named `actJobId` of the parent object. The initial value of this state variable is the `NO_JOB` constant. Any time the `runNextJob` is called, it can change its state. The developer can use also other local objects in the plug-in manager to store the current state, but in this case he needs to override the default backup functions to store the new state objects too (see Section 5.3).

First the plug-in manager sets the `actJobId` variable to the current job ID, defined in the plug-in object as static variable. Then requests all clients to run the same job (see line 20). The number of clients is stored in the `clientNum` variable of the parent class. The services of the plug-in scheduler component can be reached by the `pluginScheduler` variable, also initialized by the parent class. The `sendRunJobReq` method is putting a new event to the main event handler queue, which will send a run job request message to the given client. This method has four parameters: the ID of the given client, the ID of the given job, the parameter object if it exists, and the number of the clients will working on jobs simultaneously. The third one, the job parameter map object is not mandatory, if there is no parameter, it can be sign with `null`. If there is any parameter, then the parameter name and value pairs must be given in a `Map<String,String>` data type.

```
29
30      case DegreeDistribution.CALC_LOCAL_DEGREE_DIST:
31        TreeMap<Integer,Integer> dist = new TreeMap<Integer,Integer>();
32        //processing the results
33        for(HashMap<Integer,Integer> res :
34                    (HashMap<Integer,Integer>[]) lastResults) {
35          // processing a single distribution of a given client
36          for(Entry<Integer,Integer> deg : res.entrySet()) {
37            int degree = deg.getKey();
38            int frequency = deg.getValue();
39            if(dist.containsKey(degree)) frequency += fullDist.get(deg);
40            dist.put(degree, frequency);
41          }
42        }
43        //print out the results:
44        String distribution = "degree-frequency pairs: ";
45        for(Entry<Integer,Integer> deg : fullDist.entrySet()) {
46          distribution += "(" + deg.getKey() + "-";
47          distribution += deg.getValue() + ")";
48        }
49        logger.log(distribution, LogType.DEFAULT, this);
50
```

```
51          //this was the last job, we will not run any other
52          pluginManager.pluginFinished();
53          break;
54
55        default:
56          logger.log("invalid state in plug-in manager object",
57                                  LogType.ERROR, this);
58        }
59      }
60    }
```

When all the clients finished their jobs, the `runNextJob` method is also called. This time the `actJobId` will be not `NO_JOB`, so the plug-in manager recognizes the first job is finished. It can get the results of the last job in the array named `lastResults`. The type of this array is `Serializable[]`, and it has as many elements as the number of clients in this project. If one of the clients was not working on the last project or did not send back any result object, their results will be set as `null` in the array. The number of clients and their IDs are constant during a project.

If the plug-in manager decides there is no more job to run, it must call the `pluginFinished` method of the plug-in scheduler service, as can be seen in line 52.

5.2 Using the distributed graph database

The data objects of the distributed graph database can be reached from the client side plug-ins, more precisely from the distributed job objects. The server side plug-in manager objects can not reach the distributed database, they can get graph related information only trough the response objects of the jobs.

In the job objects, the current view of the distributed graph structure can be seen in the `network` variable. Figure 5.1. shows the class diagram of graph related interfaces can be used by the plug-in developers.

There are three kind of graph related data structure in the distributed graph database: the `Node`, the `Edge` and the `Module` objects. All of them extends the `DatabaseEntry` object, which contains some common functions, giving back the unique ID of the database object, the ID of the owner client responsible for the object and some distributed property getter and setter. In every database object, numerous property can be stored (for example the name of the object in string property, some measurements in float or integer properties, etc.). Each property has a string identifier, called `type`. The property with the given type is created when the first set (or increase) method is called for the given type. There is a special float property named `cumulatedFloatProperty` which can not be set, but increased all the time. In the merge phases when a distributed `DatabaseEntry` object is committed to its owner, all the non-local property will be overridden except the cumulated ones. In case of cumulated properties the changes are merged instead of the exact values. Currently only cumulated float properties exist.

There are relations between the three types of database objects. From the nodes, the connected edges (or the nodes connected trough them) can be listed, while from the edges, the starting and ending nodes can be reached (or the edges connected trough them). Both the nodes and edges can belong to modules with float strength values, and these belonging properties can be reached from all objects.

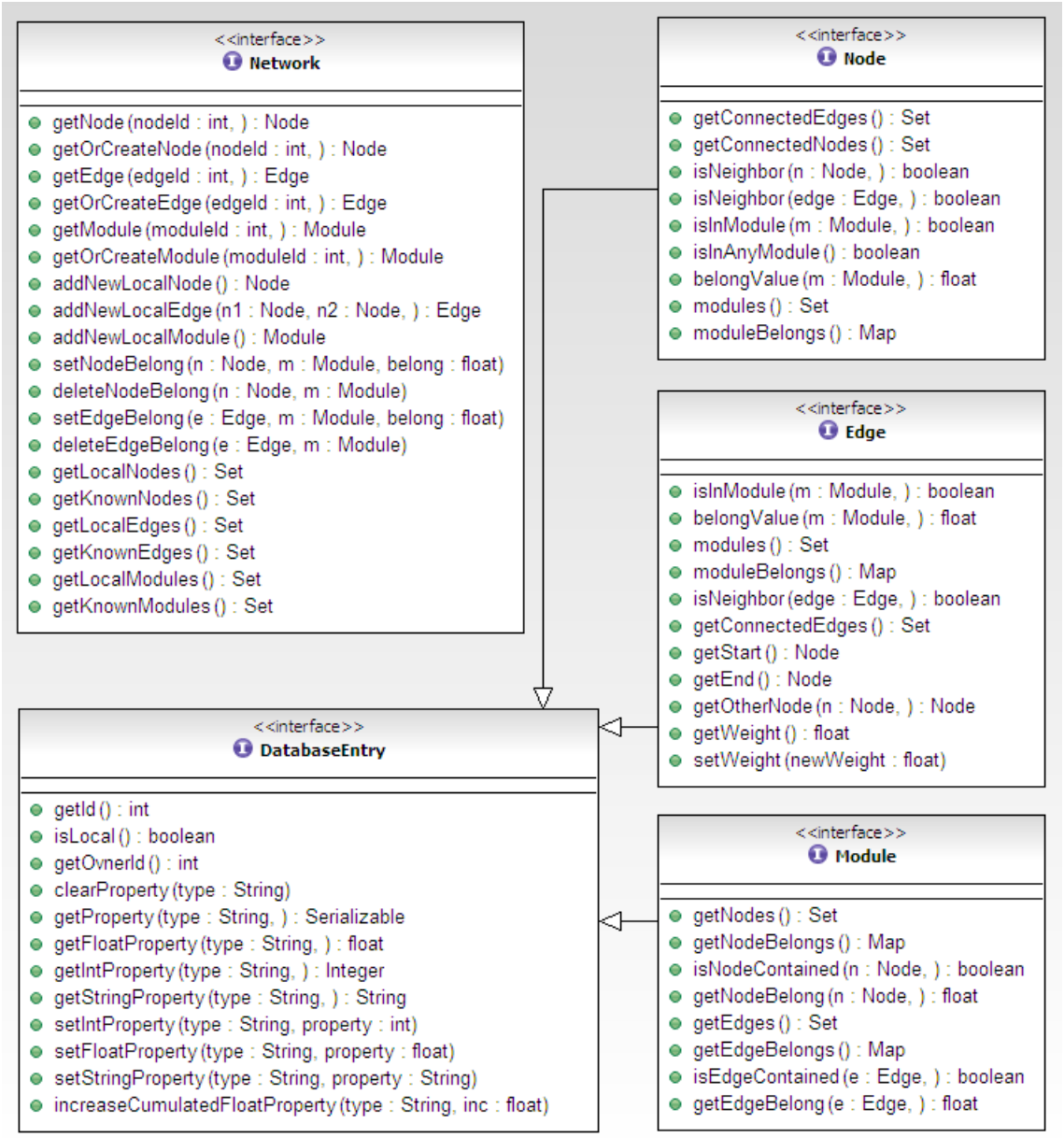

Figure 5.1: *The UML class diagram of graph db object interfaces.*

The `DatabaseEntry` objects can be reached from the `Network` object, which is responsible for managing the sets of local and known nodes, edges and modules. For example if we have a local edge object which have a local start node and a non-local end node, then the reference of the end node will be stored in the known node list. When the plug-in runs something on the non-local object in the first time, the database will request the current version of this new node object from its owner client. It is possible, when the other client sends this object, the local client will notice some new non-local edge references connecting to this new node. These new edge references will be added to the known edges list too.

The `Network` object is responsible for creating the `DatabaseEntry` objects by their unique ID with the `getOrCreate...` and without the unique ID by the `addNewLocal...` methods. The deletion of the `DatabaseEntry` objects and the assignment between modules and nodes or links are also made by the `Network` object.

5.3 Creating and restoring backups

In the following example will be shown, how to override the default backup creation and restore behavior in order to minimize the lost of computation time in case of an error. The appearance of this situation on the GUI can be seen in the screenshots of Appendix A.1 where the `linkland` plug-in was interrupted and restored.

The next distributed job sums the weights of the connected edges for each node of the network and stores it as a float property in each node.

```
1  package hu.linkgroup.paragraph.plugin.general;
2
3  import java.io.IOException;
4  import java.io.ObjectInputStream;
5  import java.io.ObjectOutputStream;
6  import java.util.ArrayList;
7
8  import hu.linkgroup.paragraph.graphdb.interfaces.Edge;
9  import hu.linkgroup.paragraph.graphdb.interfaces.Node;
10 import hu.linkgroup.paragraph.plugin.DistributedJobBase;
11 import hu.linkgroup.paragraph.services.logger.Logger.LogType;
12
13 public class SumConnectedEdges extends DistributedJobBase {
14
15   ArrayList<Node> remainingNodes = null;
16   public static String SUM_WEIGHT = "param_sumWeight";
17
18   @Override
19   public void runJob() {
20
21     if(remainingNodes==null)
22       remainingNodes = new ArrayList<Node>(network.getLocalNodes());
23     else logger.log("BACKUP FOUND", LogType.DEFAULT, this);
```

Our main concept will be to maintain an array of node references, showing still which nodes need to be deal with. This will be the `remainingNodes` variable. In the beginning of the algorithm, we initialize this array to contain all of the local nodes. When we are creating a backup, we write out this array. So if any error happens and we restore the last backup, the database will be restored with our previous calculations and we will know the list of the remaining nodes.

```
24     int i = 1;
25     while(remainingNodes.size() > 0) {
26       // creating backups in every 200 rounds
27       if(i % 200 == 0) createBackup();
28
29       Node actNode = remainingNodes.remove(0);
30       i++;
31
32       float sum = 0;
33       for(Edge e : actNode.getConnectedEdges())
34         sum += e.getWeight();
35       actNode.setFloatProperty(SUM_WEIGHT, sum);
36     }
37     createBackup(); // creates a last backup
38     myAlg.jobFinished(null);  //no result object
39   }
```

As can be seen above, we are creating a full backup after every two hundred nodes. An other backup is created by the system automatically in the beginning of the job, right before the `runJob` method would be called. We are creating an other one just in the very end of the job. If the error happens after the current client finished its job but not started the next one (for example during the merge phase), the system will be restored to this last backup and practically make no calculation at all. In the next code part we can see the overloaded backup creation function of the job class:

```
40   @Override
41   public void writeStateBackup(ObjectOutputStream oos)
42                                 throws IOException {
43     super.writeStateBackup(oos);
44     if(remainingNodes == null) oos.writeInt(0);
45     else {
46       oos.writeInt(remainingNodes.size());
47       for(Node n : remainingNodes) oos.writeInt(n.getId());
48     }
49   }
```

And the overloaded backup restoration class:

```
50   @Override
51   public void updateStateByBackup(ObjectInputStream ois)
52                 throws IOException, ClassNotFoundException {
53     super.updateStateByBackup(ois);
54     remainingNodes = new ArrayList<Node>();
55     int size = ois.readInt();
56     for(int i=0; i<size; i++)
57           remainingNodes.add(network.getNode(ois.readInt()));
58   }
59 }
```

The node references are saved and restored by their unique IDs, which remain the same even if the whole database was reloaded after an error.

Chapter 6

Performance

All the tests shown here were run in a computer laboratory at the Technical University of Budapest. All the computers had similar hardware configuration with Windows Vista host operating system, running one other Windows XP VMware image. Both the physical and virtual machines were on the same 100Mbit LAN. For all the virtual machines, one processor was allocated with 1024 MB of RAM.

From two to nine machines were used for the measurements. The server never run alone, but always together with a client and the GUI on the same machine, and all other active clients were run alone on separated computers. The run-time analysis was created by the statistics type of log events made by the ParaGrAPH framework. These logs can be found in the log files generated separately in case of each project on each machines. To help the analysis, I wrote a new little java program to process these log files and convert their information to csv format text files. This tool can be found beside the ParaGrAPH program.

6.1 Measured algorithms

For the measurements, the members of the ModuLand [14] algorithm family were used (see Section 2.4 for overview, and Appendix A.2 the exact definition of the algorithms) with a simple distributed test graph generator.

On Figure 6.1. you can see an example of the generated and clusterized test graph (the screenshot was taken in the Pajek program [4]). Each cluster is marked with different color, and the nodes are labeled with the ID of the cluster they belong at most. The generator has two parameters. Each client will create random number of local nodes between $minimumNodes$ and $2{*}minimumNodes$, and make all possible connections between them. The $interModuleRatio$ parameter determines the mean of the ratio of inter-component and intra-component links. For example, if a given client made 11 local nodes, all given local node will have 10 links to the others. And if the $interModuleRatio$ is set to 0.4, then the number of new links towards random nodes in other clients will be around 40% of its local connections, 4 in this case. On Figure 6.1., there were six clients, the $minimumNodes$ was set to 15 and the $interModuleRatio$ to 0.2. Every link created by the test graph generator has a random weight, a float value between 1 and 5. Of course in the future more tests are needed with different test graph generation methods, or using real datasets and networks.

Three members of the ModuLand algorithm family were chosen to implement as plug-ins to create overlapping clusterization of the generated networks. The plug-in named *LinkLand*

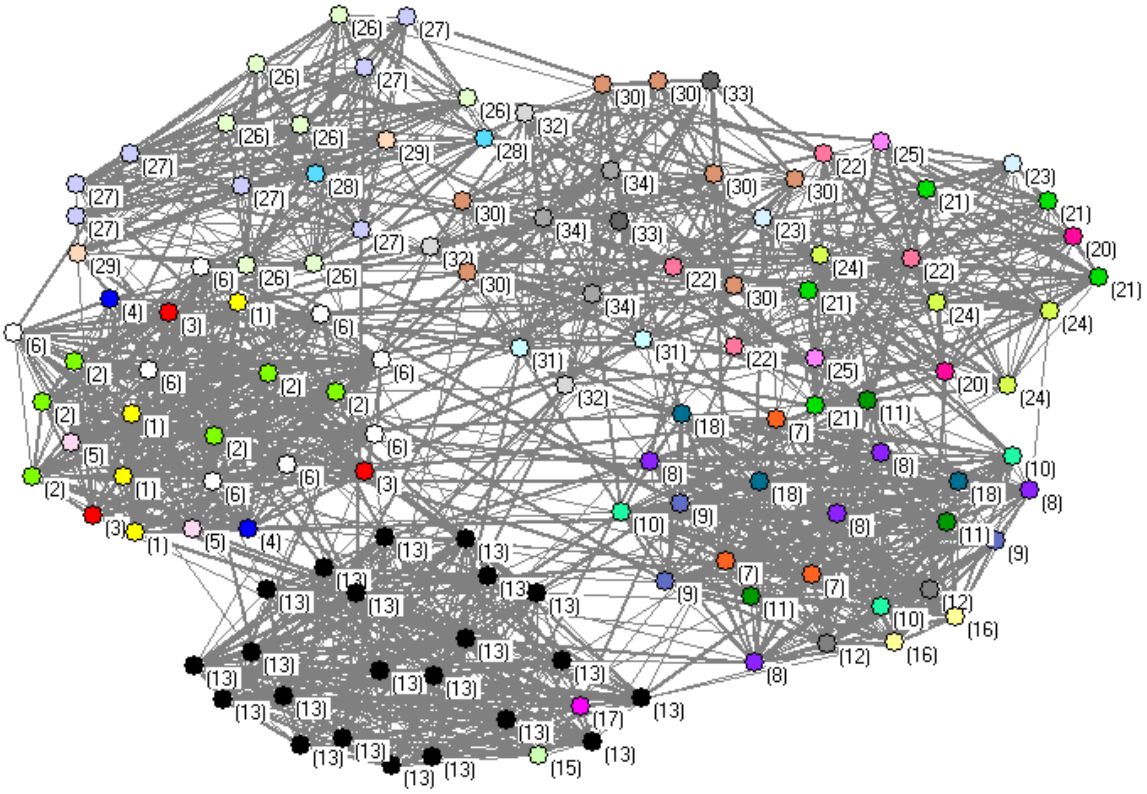

Figure 6.1: *An example network clusterized in ParaGrAPH.*

is responsible for creating the first two steps of the ModuLand clusterization. It calculates the local influence of each link to its neighborhood (Figure 2.1.A), and based on this information it creates a centrality landscape by assigning a height value to each link and node of the graph (Figure 2.1.B). The third step of the clusterization (Figure 2.1.C) is achieved by two plug-ins named *HillTopFinder* and *Proportional*. The *HillTopFinder* identifies the module centers as the local maxima on the landscape, while the *Proportional* plug-in assigns every link and node to these modules with different strength.

The plug-in *PajekExport* is responsible for exporting the graph structure and the clustering information to the text file format of the widely used Pajek[1] program [4]. It creates two files. One contains the topology of the network (the name of nodes and the parameters of links), while the other on is for storing the clustering information (for each node the id of the cluster it belongs at most).

6.2 Graph size

The first question was, how the increase of the graph affects the run-time of the *LinkLand* plug-in. The plug-in has two jobs, the first one is calculating the centrality landscape, while the second one is printing out the local height values for debugging purposes. We generated random graphs in different sizes and measured the time needed for running the first job of the *LinkLand* plug-in and the following merge phase together. In the merge phase happens the commitment of the non-local distributed database objects and the initialization of the local objects for the next job.

[1] `http://pajek.imfm.si/doku.php`

Identifying the local influence of one link in the LinkLand algorithm is structurally similar to a breadth-first search. This influence needs to be calculated in case of each link, therefore the runtime complexity of the algorithm is $O(e(n+e))$, where n is the number of nodes and e is the number of links in the network. To compare the results of the distributed *LinkLand* plug-in in the ParaGrAPH framework and the standalone algorithm, as control I used the highly optimized standalone program version, implemented in C language by the professional developers of the LinkGroup research group (the program can be downloaded from: `http://linkgroup.hu/modules.php`). Both programs were used in case of each generated network. In the Figure 6.2. can be seen the measured time values depending on the number of links.

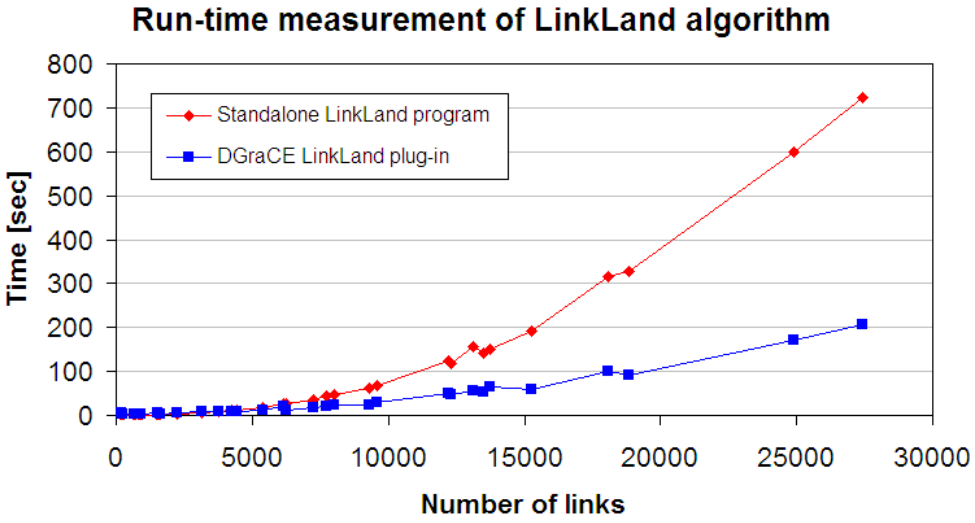

Figure 6.2: *Run-time of ParaGrAPH and standalone version of LinkLand algorithm, depending on the number of links.*

As the diagram shows, the distributed LinkLand plug-in implemented in java gives better performance in case of larger graphs than the optimized standalone C version. It is an interesting question how the ParaGrAPH version behave in the small networks (see Figure 6.3.).

In case of small graphs with links less than 4000 the optimized standalone version gives less run-time values. These graphs are really small ones and probably there is no need for running the calculations on six machines over a distributed java framework in case of problems solved around eight seconds on a single machine. However, it gives an opportunity to compare the time spent on running the distributed job and on performing the distributed database merge phase in ParaGrAPH. The sum of this two value gives the run-time of the *LinkLand* plug-in. In the job phase happens the real calculation and the periodic backup creation, while in the merge phase there is no useful calculation, only the sending of the changes on non-local database objects to their owner clients (see Section 4.2). The later is approximately the same 2.5 - 3 seconds during the whole time in case of these small networks. According to the data, considering all the runs and not just the small networks, the maximal run-time value of the merge phase is 4.328 seconds.

6.3 Scalability

In the previous section we checked how the increasing of the graph size affects the run-time of the distributed *LinkLand* plug-in. Now the size is fixed and the number of client pro-

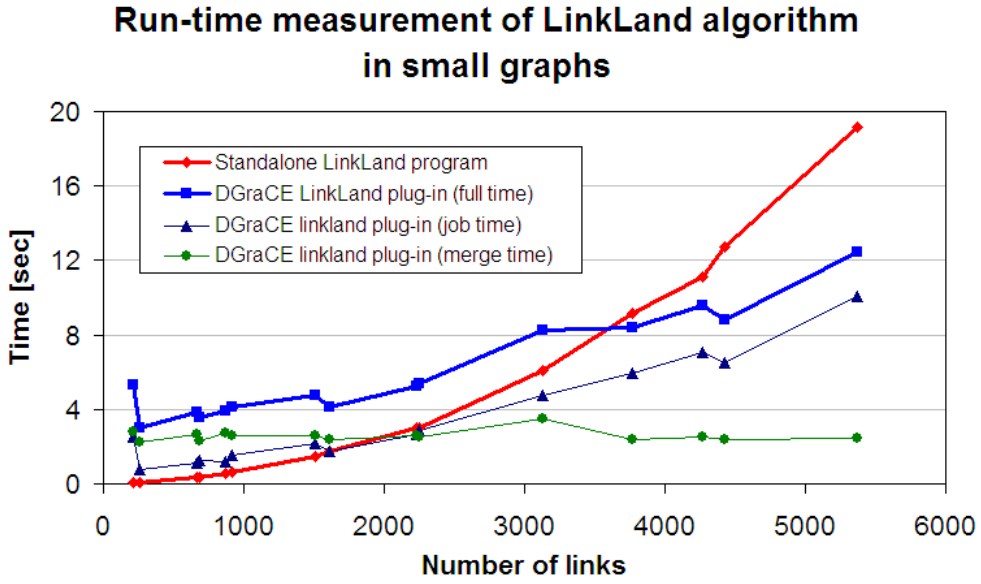

Figure 6.3: *Run-time of ParaGrAPH and standalone version of LinkLand algorithm in case of small networks.*

grams running the calculation will be the variable. To avoid the algorithmic dependencies of the given graph topology, I made several measurements on different random graphs. During the tests, from two to nine machines worked on the *LinkLand* plug-in, in every case with graphs having around 20000 links. All the tests were remade with the optimized standalone implementation of the LinkLand algorithm.

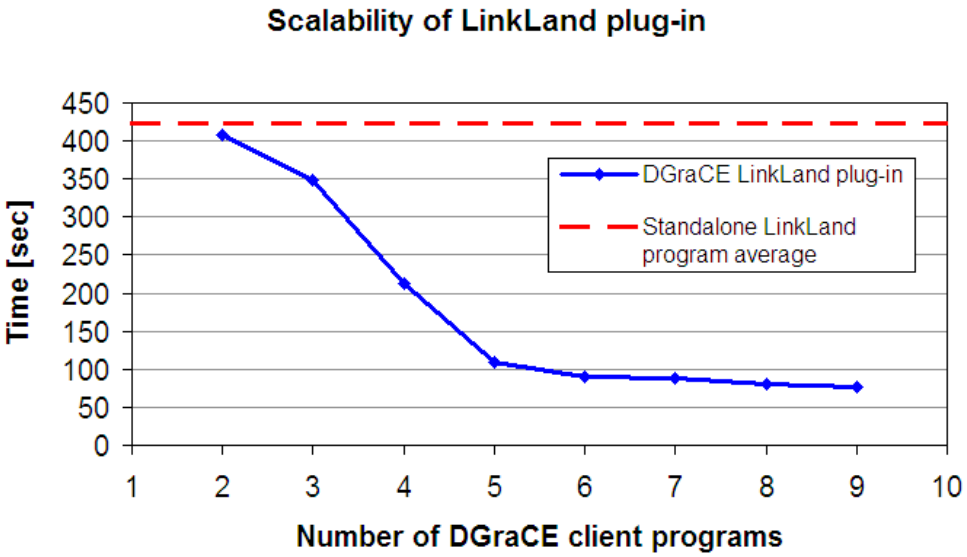

Figure 6.4: *Run-time of LinkLand plug-in depending on the number of clients.*

As Figure 6.4. shows, in this network size the standalone C implementation gives approximately the same performance as the ParaGrAPH java plug-in implementation on two hosts. If there are more than two clients running, the distributed system needs much less time. For example three machines run for 350 seconds, six machines for 90 and nine machines for 76 seconds in average. During these tests only three runs were made in case of a given client number and test graph generation algorithm has a weakness of making large differences in the number of local edges. Some client could have much more local edges

than the others, while the sum of edges still remains the same. Because of the run-time of the *LinkLand* job instance on a given client strongly depends on the number of its local edges, it is possible these average results would slightly change after making a lot more tests. However, the main characteristics of the diagram would still remain the same in case of these client numbers. In the future, other measurements are planned on different test graphs and graph sizes with other algorithms.

The scalability of the system is depending also on the memory usage. However, I was not able to measure it in such a small graph scale. The memory usage of the JVM on clients were below 40-50 MB even in case of a relatively large graphs and it contains the usage of the distributed graph database view, the JADE and ParaGrAPH frameworks such as the usage of the given plug-in. Of course the graph models of real life systems (such as a living cell or a phone network) can be much larger and the plug-ins could need much more space, so the memory usage measurement is still an important further task. It is also interesting how the Java garbage collector affects the memory usage in case of different Java versions, because for example if a new project is started on the GUI, basically all objects in the ParaGrAPH clients are recreated.

6.4 Update strategies

There is two kind of update mechanism of non-local graph database objects in the ParaGrAPH framework (see Section 4.2), can be chosen on the GUI in case of each plug-in. One method is when the update happens on each object. In this case the framework detects if the plug-in wants to read or modify a non-local database object out of date and stops the thread of the job to get the current version. The other method is to update all the database objects in some large query before the job even starts to run. The first mechanism minimizes the number of the required data objects, but can lead to a lot of time loss when the job just waits for the information. The second mechanism creates much more traffic at once in the beginning of each job and can cause a lot of unnecessary data moving, but without the drawback of waiting for data during the job. My experiences show the second way is usually much better, except in cases when the job works only on local objects.

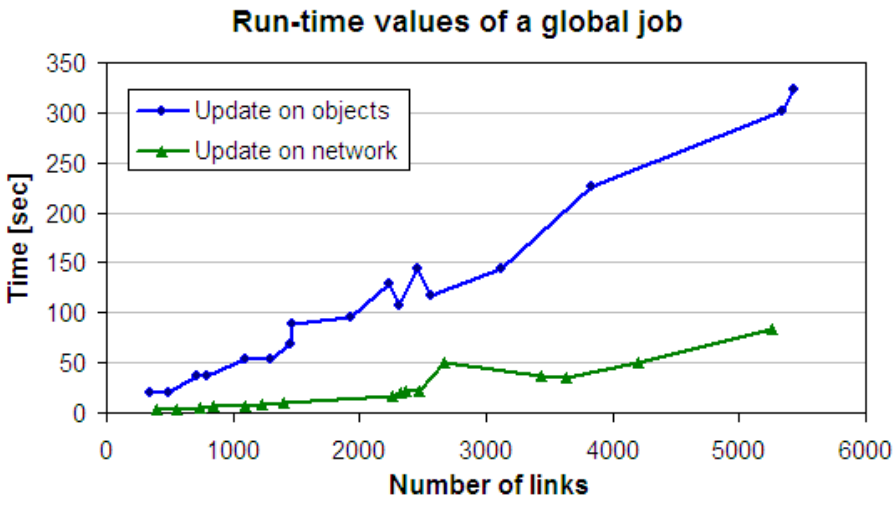

Figure 6.5: *Comparing update strategies in case of a global job (first job of the LinkLand plug-in).*

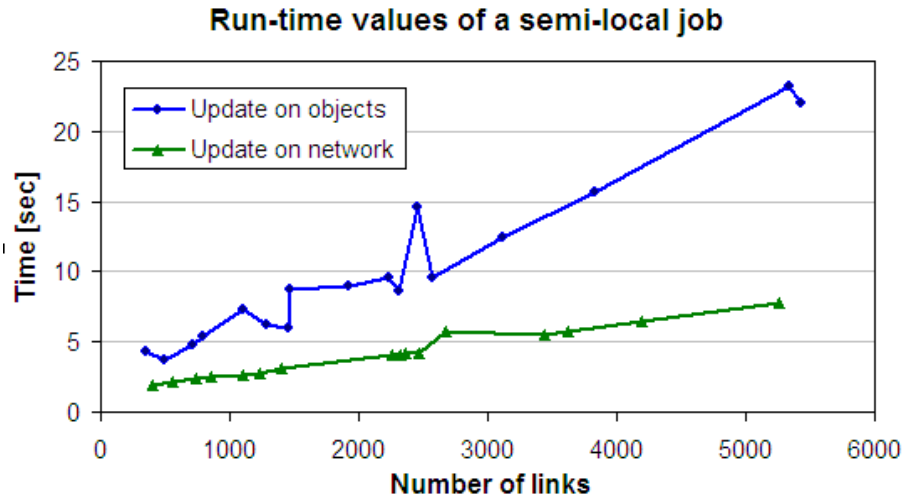

Figure 6.6: *Comparing update strategies in case of a semi-local job (second job of the HillTopFinder plug-in).*

On Figure 6.5. can be seen the comparison of the two methods in case of a job which uses almost the entire graph. In our test graphs the diameter (the maximum length of the shortest paths between any given two nodes) can not be more than three, since the graph contains full subgraphs connected to each other. The first job of the *LinkLand* plug-in was chosen to test the global behavior, because in our specific graphs it usually requires some information on the great majority of the network (in real-life networks the LinkLand algorithm behaves much more locally). The diagram shows that in case of this global job, the mechanism updates on every single object is always worse than the one updates the entire network.

There are jobs working basically on local objects and on all of their neighbors in the graph. On Figure 6.6. the second job of the *HillTopFinder* plug-in was tested, which is responsible for deciding in case of each local link if it has or has not a neighbor with more height value on the centrality landscape. In case of these semi-local jobs the single object update is closer to the other one, but still the second update mechanism is better.

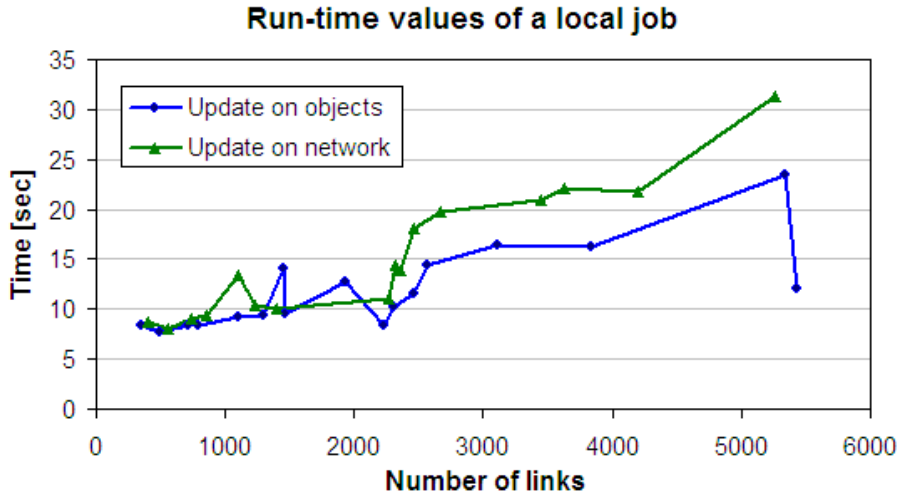

Figure 6.7: *Comparing update strategies in case of a local job (first job of the DegreeDistribution plug-in).*

Then the first and only job of the *DegreeDistribution* plug-in was tested. This job was an example in Section 5.1, which operates only on local object. The results can be seen on Figure 6.6.: in this case the job does not need update at all and of course the method which updates the whole network will take more time now.

Chapter 7

Evaluation and Conclusion

The original goal was to create a distributed graph clustering engine, to help the scientists of the field to implement and compare their algorithms easier. The Parallel Graph Algorithm Framework fulfilled this task. The most of the commonly used clustering and benchmarking methods can be realized as ParaGrAPH plug-ins, and in case of parallelizable algorithms the framework gives a scalable distributive environment. The plug-in API[1] offers many useful services to the plug-in developers, helping the implementation of graph clustering algorithms.

However, the ParaGrAPH framework is not only capable of running graph clustering algorithms, but any kind of parallel methods created for analyzing large graphs. It gives us the opportunity for creating an open source project, a framework potentially used by a large number of scientist modelling complex systems with graphs. This leads to higher level of challenges, because the ParaGrAPH framework must be enhanced with new services and more user friendly graphical and programming interfaces to achieve this goal. Personally I hope this framework can became a professional and well known tool for scientists and will assist to prove or question many theories about complex systems.

7.1 Further improvements

During the development of the current version of the Parallel Graph Algorithm Framework, a lot of idea came up as possible further enhancement. In the near future the framework will be uploaded to the SourceForge[2] open source project library and hopefully reach some attention from users and developers as well. In the following lists some of the possible changes and potential new functionalities are gathered.

ParaGrAPH development related ideas:

- Some unit test mechanism and formal validation of state machines and message protocols would be highly needed even now, and essential if the project become larger and more complex.
- Longer tests (running days) on larger graphs should be made to estimate the practical limits of the framework, for example investigte the memory usage.

[1] Application Programming Interface
[2] `http://sourceforge.net`

GUI related ideas:

- The GUI needs to be more user friendly, and enriched with many new functionalities.
- It would be useful to have other management interfaces as well (like command line interface or file interface to read projects based on XML descriptors) especially for supporting massive number of runnings. Thanks to the layer structure of the current GUI, this can be easily achieved.
- The ParaGrAPH could support the development of plug-in specific GUI modules.
- The ParaGrAPH should give a new interface to the plug-ins to show their progress and status on the GUI.

Distribution and availability related ideas:

- The compatibility with the current JADE version 4.0 (released on 04/20/2010) would be important. It is also possible, the new version makes easier some other enhancement of ParaGrAPH.
- For increase the reliability of ParaGrAPH, some advanced JADE techniques or other non-JADE solutions need to be considered.
- Since some of the graph algorithms demand huge computing capacity, it would be important to help the plug-in developers to reach advanced and optimized third party programs (like R or MathLab), and even give the support to run the ParaGrAPH on GPU environment.
- In some cases when the algorithm is operating on local graph parts it could be important to create a database distribution optimization to ensure the given client is not responsible for a random set of nodes and links, but a locally connected region of the graph. It could minimize the number of updated and committed database objects.
- The framework should support the creation of Amazon Machine Image contains the whole ParaGrAPH system together with the input files and plug-ins defined by the user. The created image could be deployed on Amazon Elastic Compute Cloud[3] in as many copies as many are optimal for the distributed computing.
- The parameter of the maximal number of clients can be used in a plug-in should be freely changeable on the GUI. (now it is a static integer variable set to 20)

Plug-in related ideas:

- It would be important to implement benchmarking algorithms to help the comparison of different graph clustering algorithms.
- New I/O plug-ins are needed to support the import and export in well known graph data formats (like CSV, XLS or the specific data formats used in Pajek or Citoscape programs).

[3] Amason EC2: http://aws.amazon.com/ec2

- The ParaGrAPH graph API interface should be enriched with functions to handle directed links, hyper-graphs and other graph structures.
- The versioning of different plug-ins would also be important.
- The distributed graph database should be capable to store not only one, but many graphs at one time.
- A new functionality in the plug-in service would be to create a plug-in distribution solution to upload the plug-ins on the GUI and spread it to the whole system.
- More detailed tutorials and examples should be made about plug-in development. And more, commonly used graph algorithms should be created.
- A basic unit test should be run with simple test graphs on the uploaded plug-ins and plug-in management objects to find the most simple errors (or even security threats).
- An online storage of ParaGrAPH plug-ins should be developed where the plug-in developers could share their new plug-ins.

Acknowledgements

I would like to thank András Kövi, my supervisor at the Department of Measurement and Information Systems of Budapest University of Technology and Economics, for the guidance during the project. I also received a lot of help from the LinkGroup research group. Especially I would like to thank professor Péter Csermely, the leader of the group, István Kovács, who worked out the idea of the original ModuLand method family [14], and Robin Palotai and Gábor Szuromi who helped in its implementation. I also would like to thank Melinda Bekő the final extensive linguistic review of this thesis.

Appendix

A.1 Using ParaGrAPH

The source code, the binary version and all the required library can be found in the project directory (enclosed to the thesis on CD). The following list shows the description of the main files and directories:

- *paragraph/src*: source code of ParaGrAPH system
- *paragraph/bin*: binary class files of ParaGrAPH system
- *paragraph/lib*: required libraries in jar files (all contained jar file get into the classpath by ant)
- *paragraph/doc*: documentation (currently this thesis)
- *paragraph/paragraph.jar*: the jar file built by ant contains the class files of the bin directory
- *paragraph/build.xml*: ant xml file defines the ant targets

One may want to run more than one client on a single machine. Since the backup files have the same name in case of all clients, it is a good practice to create different directories for each client running locally and copy the ParaGrAPH to each of them.

The `paragraph.jar` can be created by the `ant dist` command, and the server can be started with the `ant run_server` command, generating the following console output:

```
Buildfile: build.xml

init:

run_server:
     [java] May 8, 2010 12:09:35 PM jade.core.Runtime beginContainer
     [java] INFO: -----------------------------------
     [java]     This is JADE 3.7 - revision 6154 of 2009/07/01 17:34:15
     [java]     downloaded in Open Source, under LGPL restrictions,
     [java]     at http://jade.tilab.com/
     [java] -----------------------------------
     [java] May 8, 2010 12:09:36 PM jade.core.BaseService init
     [java] INFO: Service jade.core.management.AgentManagement
        initialized
     [java] May 8, 2010 12:09:36 PM jade.core.BaseService init
     [java] INFO: Service jade.core.messaging.Messaging initialized
```

```
     [java] May 8, 2010 12:09:36 PM jade.core.BaseService init
     [java] INFO: Service jade.core.mobility.AgentMobility initialized
     [java] May 8, 2010 12:09:36 PM jade.core.BaseService init
     [java] INFO: Service jade.core.event.Notification initialized
     [java] May 8, 2010 12:09:36 PM jade.core.messaging.
        MessagingService clearCachedSlice
     [java] INFO: Clearing cache
     [java] May 8, 2010 12:09:36 PM jade.mtp.http.HTTPServer <init>
     [java] INFO: HTTP-MTP Using XML parser com.sun.org.apache.xerces.
        internal.jaxp.SAXParserImpl$JAXPSAXParser
     [java] May 8, 2010 12:09:36 PM jade.core.messaging.
        MessagingService boot
     [java] INFO: MTP addresses:
     [java] http://gondor:7778/acc
     [java] May 8, 2010 12:09:36 PM jade.core.AgentContainerImpl
        joinPlatform
     [java] INFO: ----------------------------------------
     [java] Agent container Main-Container@gondor is ready.
     [java] ----------------------------------------
     [java] [CLASS]: hu.linkgroup.paragraph.control.ControlServerAgent
     [java] [AGENT]: server
     [java] [TYPE]: DEBUG LOG
     [java] [DATE]: Sat May 08 12:09:36 CEST 2010 (1273313376447)
     [java] [TEXT]: Control service started, server is running
     [java]
```

The command `ant run_manager` starts the GUI. If the server is not running on the local machine, its host name and port must be defined. For the usage please see the `ant help`. On the Figure A.1. can be seen the GUI.

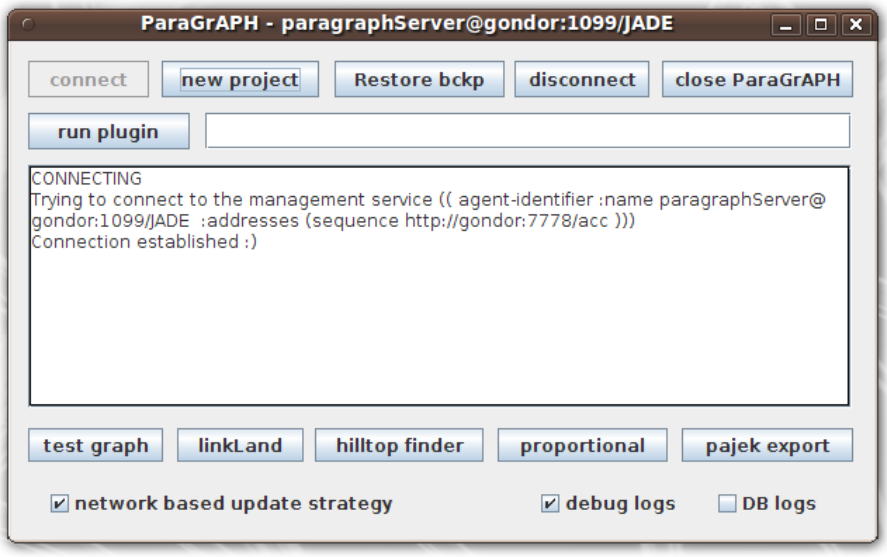

Figure A.1: *Connecting to the ParaGrAPH server.*

With the `connect` button we can connect to the ParaGrAPH management service. The `new project` creates a new empty project, while the `restore bckp` rolls back the system to the last saved state. The `disconnect` button disconnects the GUI from the server. The `close` button is shutting down every client and server programs.

The `run plugin` sends the request of running new plug-in to the server. The parameters can be written to the empty field next to the button. The *<parameter name>:<value>* pairs are separated by commas. The two mandatory parameters are the `pluginName` and `packagePrefix`. To simplify the tests, five buttons were created to fill out this parameter line. However, much more user friendly solution is needed in the future to run different plug-ins.

There are three checkboxes in the bottom of the window. Here can be altered the log level of the project and the update mechanism used during the given plug-in. If the later is selected, than a new parameter named `updateNetwork` with value *true* will be inserted to the parameter line, if a plug-in button is pressed. This parameter needs to be given if the network based update mechanism is wanted by the user.

Figure A.2: *Running linkland plug-in.*

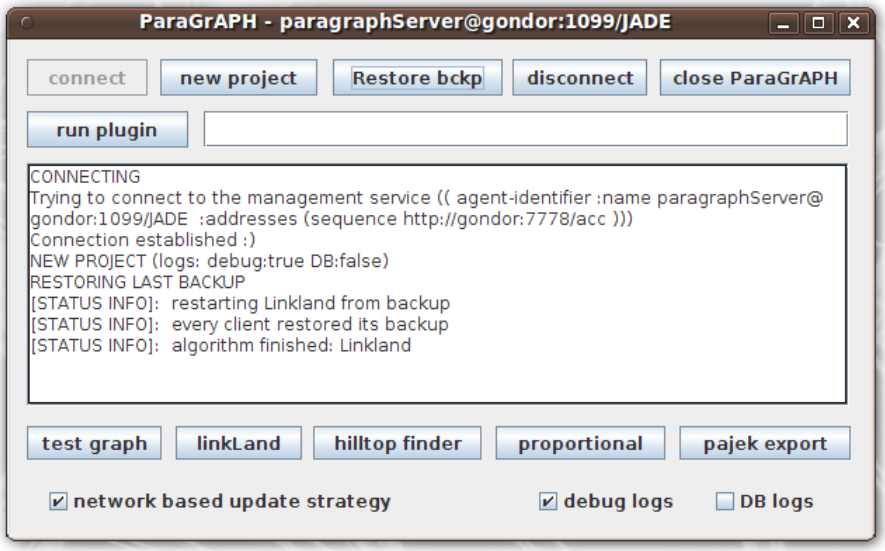

Figure A.3: *Restoring linkland plug-in.*

Figures A.2. and A.3. show the situation, when an error happens during the `linkland` plug-in and the system needs to be restored from the last backup. In case of a problem

(for example exception during the running of the plug-in) the system needs to be restarted manually and by the button `restore bckp`, the framework restores its state, and continues the running of the last plug-in, as can be seen on Figure A.3..

The developer of a new plug-in needs to ensure, his plug-in related classes are in the classpath on each JVM. It can be done for example if he copies his java files to the package hierarcy in the `src` directory, or creates a new jar file in the `lib` folder. At least two classes are needed, one plug-in class on client side and one plug-in manager class on server side. In both cases the two classes need to be in the same package somewhere inside the `hu.linkgroup.paragraph.plugin`.

A.2 Implemented members of the ModuLand algorithm family

The brief descriptions of the main steps of the ModuLand method family can be found in Section 2.4. Here we describe the two algorithms were developed as a distributed ParaGrAPH plug-in. The following two sections are copyed from the supplementary discussions of article [14].

A.2.1 LinkLand centrality landscape calculation

The LinkLand method is a fast, but approximating method for the determination of the community heaps in weighted, undirected networks. Here the community heap belonging to the starting element or link is determined by a network walk. The starting link (and later its growing community heap) is extended by those neighboring elements and their links linking them to the existing community heap and also to each other, which will at least not decrease the community heap-threshold of the existing community heap; the community heap is ready once such extension is no longer possible. The community heap-threshold of the LinkLand method is defined as the summarized weight of the links in the community heap divided by the number of nodes in the heap.

The following pseudo code shows how the LinkLand community heap construction method selects the members of the community heap in case of a given starting link of the network. The definition of the important variables used in the algorithm:

- *startLink*: the starting link of the actual community heap.
- *heapNodeList*: elements of the community heap (initially empty).
- *heapLinkList*: links of the community heap (initially empty).
- *tempList*: elements to be added to the community heap in the next round.
- *actualHeapThreshold*: sum of the weight of all links in heapLinkList divided by the number of elements in heapNodeList.

The pseudo code of LinkLand algorithm:

```
clear tempList
add the two end-elements of startLink to tempList
while tempList is not empty {
```

```
    add all elements of tempList to heapNodeList.
    for each link e connected to any elements of tempList {
      if endpoints of e are already in heapNodeList {
        add e to heapLinkList
      }
    }

    clear tempList
    recalculate actualHeapThreshold
    maxNewHeapThreshold := actualHeapThreshold

    for each each element n not in heapNodeList but having non-zero
        links lks. with an endpoint in heapNodeList {
      newHeapThreshold := sum of the weight of all links in
          heapLinkList + sum weight of link in lks
      newHeapThreshold := newHeapThreshold / (number of elements in
          heapNodeList + 1)
      if newHeapThreshold > maxNewHeapThreshold {
        clear tempList
        maxNewHeapThreshold := newHeapThreshold
      }
      if newHeapThreshold = maxNewHeapThreshold {
        add n to tempList
      }
    }
  }
```

In the end of the LinkLand algorithm we find the links and elements of the community heap in the heapLinkList and heapNodeList, respectively. Identifying the community heap of one link in the LinkLand algorithm is structurally similar to a breadth-first search, therefore the runtime complexity of the algorithm is $O(e(n+e))$, where n is the number of nodes and e is the number of links in the network. However in practice the algorithm is very fast as a community heap of any given link rarely covers the whole network.

A.2.2 Proportional module assignement method

In the Proportional module membership assignment method links of the network are assigned to modules of their non-lower neighboring links in the proportion of the absolute community landscape height of the respective neighboring links. The links having no higher neighboring link are assigned with full height to the respective modules defined by themselves.

At the start of the Proportional module membership assignment method all links are marked as unassigned. After this, multiple rounds of link-assignments are performed: in each round, links are assigned to modules based on the assignment of previously assigned links. In each round, we descend to next slice of links, starting from the top community landscape slice, where a community landscape slice is formed by all links having the same community landscape height.

Here we describe the steps of a single round of the Proportional module membership assignment method:

- The first step: each of the hill-tops/highlands of the community landscape (connected components of the actual community landscape slice without higher neighboring

links) becomes a new module-core. Each link of all these connected components is assigned to its respective new module with an assignment-strength of its full community landscape height.

- In consecutive steps, unassigned links of the community landscape slice having at least one neighboring link already assigned to the growing modules, are assigned to modules proportional to the assignment-strength of their neighbors already assigned to existing modules. In such a step, links assigned in the current step are not considered as 'assigned neighbors' during the respective step. The step described here is repeated until there are any unassigned links remaining in the actual community landscape slice. Once all links of the actual community landscape slice have already been assigned to modules, the round is over and the next round begins, unless there are no more (lower) community landscape slices left, in which case the whole assignment procedure ends.

As an outcome of the Proportional module membership assignment process, for each link the sum of the assignment-strength values of the given link to the different modules is equal to the community landscape height of that link.

The runtime complexity of the Proportional module membership assignment method is $O(edm)$, where e is the number of links of the network, d is the average degree of nodes and m is the number of modules. Assuming practically that the average degree is bounded by a constant and that the number of modules is not more than the number of nodes, the runtime complexity is $O(n^3)$.

Bibliography

[1] G. Agarwal and D. Kempe. Modularity-maximizing network communities via mathematical programming. *Eur. Phys. J. B*, 66:409–418, 2008.

[2] R. D. Alba. A graph-theoretic definition of a sociometric clique. *Journal of Mathematical Sociology*, 3:113–126, 1973.

[3] R. D. Alba and G. Moore. Elite social circles. *Sociol. Meth. Res.*, pages 167–188, 1978.

[4] V. Batagelj and A. Mrvar. Pajek - analysis and visualization of large networks. In M. Junger and P. Mutzel, editors, *Graph Drawing Software*, pages 77–103. Springer, Berlin, 2003.

[5] J. W. Berry, B. Hendrickson, R. A. LaViolette, and C. A. Phillips. Tolerating the community detection resolution limit with edge weighting. *ArXiv e-prints*, 0903.1072, mar 2009.

[6] S. P. Borgatti, M. G. Everett, and P. R. Shirey. Ls sets, lambda sets and other cohesive subsets. *Social Networks*, 12:337–358, 1990.

[7] G. W. Flake, S. Lawrence, L. Giles, and F. M. Coetze. Self-organization and identification of web-communities. *IEEE Computer*, 35:66–71, 2002.

[8] S. Fortunato. Community detection in graphs. *Phys. Rep.*, 486:75–174, 2010.

[9] S. Fortunato and C. Castellano. Community structure in graphs. *ArXiv e-prints*, 0712.2716, dec 2007.

[10] Santo Fortunato and Marc Barthelemy. Resolution limit in community detection. *Proc. Natl. Acad. Sci. USA*, 104:36–41, 2007.

[11] M Girvan and M. E. J. Newman. Community structure in social and biological networks. *Proc Natl Acad Sci USA*, 99(12):7821–6, 2002.

[12] Roger Guimerà and Luís A. Nunes Amaral. Functional cartography of complex metabolic networks. *Nature*, 433(7028):895–900, Feb 2005.

[13] M. Kitsak, M. Riccaboni, S. Havlin, F. Pammolli, and H. E. Stanley. Structure of business firm networks and scale-free models. *ArXiv e-prints*, 0810.5514, oct 2008.

[14] István A. Kovács, R. Palotai, Máté S. Szalay, and Péter Csermely. Community landscapes: an integrative approach to determine overlapping network module hierarchy, identify key nodes and predict network dynamics. *ArXiv e-prints*, 0912.0161, 2009.

[15] J. M. Kumpula, J. Saramäki, K. Kaski, and J. Kertész. Limited resolution in complex network community detection with potts model approach. *Eur. Phys. J. B*, 56:41–45, 2007.

[16] A. Lancichinetti, S. Fortunato, and F. Radicchi. Benchmark graphs for testing community detection algorithms. *Physical Review E*, 78(4):46110, 2008.

[17] Andrea Lancichinetti and Santo Fortunato. Benchmarks for testing community detection algorithms on directed and weighted graphs with overlapping communities. *Phys. Rev. E*, 80(1):016118, Jul 2009.

[18] S. Lehmann, M. Schwartz, and L. K. Hansen. Deterministic modularity optimization. *Eur. Phys. J. B*, 60:83–88, 2007.

[19] E. A. Leicht, P. Holme, and M. E. J. Newman. Modularity-maximizing network communities via mathematical programming. *Phys. Rev. E*, 73:026120, 2006.

[20] R. D. Luce. Connectivity and generalized cliques in sociometric group structure. *Psychometrika*, 15:169–190, 1950.

[21] C. P. Massen and J. P. K. Doye. Identifying communities within energy landscapes. *Phys. Rev. E*, 71:046101, 2005.

[22] D. W. Matula. K-components, clusters and slicings in graphs. *SIAM J. Appl. Math.*, 22:459–480, 1972.

[23] Robert J. Mokken. Cliques, clubs and clans. *Quality and Quantity*, 13(2):161–173, April 1979.

[24] M. E. J. Newman. Detecting community structure in networks. *The European Physical Journal B*, 38:321–330, 2004.

[25] M. E. J. Newman. Fast algorithm for detecting community structure in networks. *Phys. Rev. E*, 69:066133, 2004.

[26] M. E. J. Newman. Detecting community structure in networks. *Phys Rev E*, 74, 2006.

[27] M. E. J. Newman and M. Girvan. Detecting community structure in networks. *Phys. Rev. E*, 69:026113, 2004.

[28] Gergely Palla, Imre Derenyi, Illes Farkas, and Tamas Vicsek. Uncovering the overlapping community structure of complex networks in nature and society. *Nature*, 435(7043):814–818, June 2005.

[29] Béjar J. Pujol, J. M. and J. Delgado. Clustering algorithm for determining community structure in large networks. *Phys. Rev. E*, 74:016107, 2006.

[30] M. Rosvall and C. T. Bergstom. An information-theoretic framework for resolving community structure in complex networks. *Proc. Natl. Acad. Sci. USA*, 104:7327–7331, 2007.

[31] M. Rosvall and C. T. Bergstom. Maps of random walks reveal community structure in complex networks. *Proc. Natl. Acad. Sci. USA*, 105:1118–1123, 2008.

[32] P. Schuetz and A. Caflish. Efficient modularity optimization: multi-step greedy algorithm and vertex mover refinement. *Phys. Rev. E*, 77:046112, 2008.

[33] Stephen B. Seidman. Network structure and minimum degree. *Social Networks*, 5:269–287, 1983.

[34] Stephen B. Seidman and Brian L. Foster. A graph-theoretic generalization of the clique concept. *Journal of Mathematical Sociology*, 6:139–154, 1978.

[35] G. Tibély and J. Kertész. On the equivalence of the label propagation method on community detection and a potts model approach. *Physica A*, 387:4982–4984, 2008.

[36] Stanley Wasserman and Katherine Faust. *Social Network Analysis: Methods and Applications*. Cambridge University Press, 1994.