Overlapping Non-dedicated Clusters Architecture

by

Martin Štava

A thesis submitted to the Faculty of Information Technology, Czech Technical University in Prague, in partial fulfilment of the requirements for the degree of Doctor.

PhD programme: Informatics

Prague, November 2012
Abstract and contributions

This doctoral thesis deals with design and implementation of a secure peer to peer clustering system. Our work is motivated by a known fact that most of the workstations are heavily underutilized and they stay nearly idle for a large part of a day. The main goal of our system is to allow efficient and secure harvesting of those unused computing resources on local networks.

The observation about workstations under utilization is not new and so there is a large variety of systems with similar goals. Among various used approaches, the single system image clusters are unique in their ability to hide system complexity from cluster users. Traditionally, single system image clusters assumed deployment in a secure trusted environment and did not provide any security mechanisms. Authors of Clondike system were the first to introduce a concept of security into the single system image clusters world.

Despite being very innovative, the Clondike system had several deficiencies that limited its usability. The main problem was a partially centralized system architecture, that relied on a single master node to control the cluster. The node was both single point of failure and a performance bottleneck. The security mechanisms were only briefly sketched and no real scheduling algorithm was proposed. In addition, all cluster services were centralized.

In our work, we take a new look on a secure single system image cluster architecture. We address the above mentioned deficiencies of the original Clondike system and demonstrate feasibility of our design proposals by implementing them as an extension to the system.

In particular, the main contributions of this doctoral thesis are as follows:

1. A proposal and implementation of a fully decentralized secure peer to peer single system image clustering solution, that does not contain any single point of failure points or performance bottlenecks in form of centralized components.

2. A security system inspired by trust management systems, tailored for usage in peer to peer clusters, was proposed and its core was implemented for the Clondike system. A similar system could be used for any other peer to peer clustering solution.

3. A flexible scheduling framework, that is extremely easy for experiments and even non expert users can create their own customized scheduling strategies. A simple, but robust, scheduling algorithm was implemented and its performance observed through measurements.

Keywords:
Cluster computing, Grid computing, peer to peer networks, security, single system image.
Acknowledgements

First of all, I would like to express my gratitude to my dissertation thesis supervisor, Prof. Pavel Tvrdík. Since beginning of my studies, he has been pushing me in the right direction.

I would like to thank to Ing. Martin Kačer PhD. and Ing. J. Čapek for starting an interesting research that I took over and also to Ing. Josef Gattermayer for helping me with measurements in computer laboratory.

Special thanks go to the staff of the Department of Computer Systems, who maintained a pleasant and flexible environment for my research. I would like to express special thanks to the department management for providing most of the funding for my research.

My research has also been partially supported by the Ministry of Education, Youth, and Sport of the Czech Republic under research program MSM 6840770014, and by the Czech Science Foundation as project No. 201/06/1039. Part of the research was also funded by the Czech Grant Agency GACR under grant No. 102/06/0943.

Finally, my greatest thanks go to my family members, and especially to my wife, who had to suffer countless weekends and evenings with me sitting just in front of computer without any interaction with the outer world.
Dedication

To my wife for her endless patience.
## Contents

<table>
<thead>
<tr>
<th>List of Figures</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Tables</td>
<td>xiv</td>
</tr>
</tbody>
</table>

### 1 Introduction
1.1 Motivation 1
1.2 Vision 3
1.3 Contributions of the Thesis 4
1.4 Authorship of the Work 5
1.5 Scope of the Work 7
1.6 Structure of the Thesis 7

### 2 Background and State-of-the-Art
2.1 Theoretical Background 9
  2.1.1 Operating systems 9
    2.1.1.1 Programs 10
    2.1.1.2 Operating modes 10
    2.1.1.3 Virtualization 11
  2.1.2 Linux OS 12
    2.1.2.1 Memory management 12
    2.1.2.2 System calls 13
    2.1.2.3 Processes and threads 13
    2.1.2.4 Filesystems 14
    2.1.2.5 Pipes 15
    2.1.2.6 Executables 16
    2.1.2.7 Signals 16
    2.1.2.8 Networking 16
    2.1.2.9 Access control 17
    2.1.2.10 Modules 18
  2.1.3 Distributed computing 18
    2.1.3.1 Computers and nodes 18
    2.1.3.2 Networking 19
    2.1.3.3 Goals 20
## CONTENTS

2.1.3.4 Distributed system architectures ........................... 21
2.1.3.5 Single system image ........................................ 22
2.1.3.6 Data sharing ............................................... 23
2.1.3.7 Process migration ........................................... 24
2.1.3.8 User roles .................................................. 24

2.2 Previous Results and Related Work .............................. 25
  2.2.1 openMosix/Mosix ............................................. 25
  2.2.2 OpenSSI ...................................................... 26
  2.2.3 Kerrighed ..................................................... 26
  2.2.4 NOW .......................................................... 26
  2.2.5 Condor ....................................................... 27
  2.2.6 Beowulf clusters ............................................. 27
  2.2.7 Globus toolkit ............................................... 28
  2.2.8 XtreemOS ..................................................... 28
  2.2.9 OurGrid ....................................................... 29
  2.2.10 Virtual organization clusters ................................ 29
  2.2.11 BonjourGrid ................................................. 29
  2.2.12 Plan 9 ........................................................ 30
  2.2.13 Clondike ..................................................... 30

3 System Architecture .................................................. 31
  3.1 Full decentralization .......................................... 33
  3.2 Process migration ............................................... 34
  3.3 Filesystem ....................................................... 35
    3.3.1 Caching ...................................................... 36
  3.4 Security .......................................................... 37
    3.4.1 Identity ...................................................... 38
    3.4.2 Access control and certificates .......................... 39
    3.4.3 Migration security and stigmata ......................... 47
    3.4.4 Stigmata and caching ..................................... 48
    3.4.5 Scheduling with stigmata ................................ 49
    3.4.6 Filesystem security ....................................... 50
    3.4.7 Data transfer security ................................... 50
    3.4.8 Denial of service .......................................... 51
    3.4.9 Handling unknown nodes ................................... 51
  3.5 Scheduling ....................................................... 52
    3.5.1 Used algorithm ............................................. 53
  3.6 Network interconnection ......................................... 55
    3.6.1 LAN channel security ..................................... 56
    3.6.2 WAN channel security ..................................... 57
  3.7 Classifications and predictions ................................ 58
  3.8 Process trace dumper ............................................ 58
# CONTENTS

## 4 Implementation

- 4.1 Overview .......................................................... 59
- 4.2 User mode director ............................................. 60
- 4.3 Communication channels .................................... 62
- 4.4 Migration mechanisms ........................................ 63
- 4.5 Monitoring and information dissemination .............. 67
- 4.6 Proxy filesystem ............................................... 67
- 4.7 Scheduling ........................................................ 68
- 4.8 Security .......................................................... 70
- 4.9 Filesystem integration ....................................... 71
- 4.10 Caches .......................................................... 73
- 4.11 Mixing 32 bit and 64 bit computers ...................... 74

## 5 Experimental Evaluation

- 5.1 Measurements framework .................................. 77
- 5.2 Measurements environment ................................ 79
- 5.3 Execution measurements in homogeneous cluster .... 79
  - 5.3.1 Mandelbrot set calculation ........................... 80
  - 5.3.2 Kernel compilation .................................... 84
- 5.4 Measurement on heterogeneous clusters ............... 91
- 5.5 Predictions accuracy ......................................... 93

## 6 Conclusions

- 6.1 Summary .......................................................... 95
- 6.2 Contributions of the Thesis ................................ 96
- 6.3 Future Work ...................................................... 96
  - 6.3.1 Filesystem ............................................... 97
  - 6.3.2 Scheduling ............................................... 97
  - 6.3.3 More powerful migration techniques ............... 98
  - 6.3.4 Introduce reputation into security ................. 99
  - 6.3.5 Interconnection of large networks ................. 99
  - 6.3.6 MPI integration ....................................... 99

**Bibliography** .................................................. 101

**Publications of the Author** .................................. 109

**A Cluster APIs exposed by kernel** ......................... 111
List of Figures

3.1 A schematic figure of a dedicated cluster. All nodes are equal and form a single cluster. .......................................................... 32
3.2 A schematic figure of a non-dedicated cluster. There is a distinguished node forming the cluster and other nodes are ordinary workstations. The workstations run a nested virtual environment that could be used by the cluster. .......................................................... 33
3.3 A schematic figure of an overlapping non-dedicated cluster. Each node forms its own cluster and acts as a core of that cluster. Each node acting as a core node can use other nodes as detached nodes of its cluster. Each of the nodes acting as a detached node hosts an internal nested virtual environment for each other connected core node. .......................................................... 34
3.4 First, node A grants some privileges to node B, including a delegation rights. Then node B grants a subset of those rules to node C. It provides both the certificate with granting of the privileges and a certificate proving that it is allowed to grant such rules. .......................................................... 41
3.5 First, owner of nodes A, B, and C delegates all permissions from nodes A and B to node C. Then, from node C he makes a group G1 containing both A and B. From then on, he can grant permissions to the whole group at once. 42
3.6 When the node C grants some privileges to operate on A to the node D, the node A does not directly know the certificate issued by C is valid. It can know this only if it sees a delegation certificate from B granting permission to the group G and also a membership certificate issued by B assigning the node C to the group G. .......................................................... 44
3.7 Full lines represent compulsory propagation as required by the protocol. Dotted lines represent optional certificates propagation if the back propagation is enabled. .......................................................... 45
4.1 High level layers of implementation ........................................ 60
4.2 Basic view of director components ........................................ 61
4.3 The most important communication channels used by the system 62
4.4 The non-preemptive migration can be started only when a process invokes the `exec` system call. For each eligible process, the `exec` syscall sends a netlink message to the director to check if the process should be executed locally or migrated away. The scheduler is consulted and replies back using the netlink channel either saying the process should stay on the core node, or a node where to migrate the newly executed process. If the scheduler decides that the process should migrate, the process will not execute the `exec` system call locally, but instead it will initiate a non-preemptive process migration.

4.5 The preemptive migration is initiated by the scheduler that sends a message to the kernel using the pseudo filesystem channel. The kernel in turn sends a signal to the process that is being migrated and when the process picks up the signal, it initiates the migration process on itself.

4.6 After the migration process is initiated, the flow of events is mostly the same for the preemptive and non-preemptive migration. The current application is frozen and a checkpoint is made. After the checkpoint creation is completed, the kernel informs a peer node kernel that the checkpoint is ready. The peer node kernel consults its director, whether it is willing to accept the process or not. If the director rejects the migration, the core node is notified back and the migration is aborted and the process runs locally. Otherwise, the migration proceeds.

4.7 Flow of events in case a security check is initiated as a result of invoking a syscall by an application. Responses use the same channels in the opposite direction.

4.8 Flow of events in case a security check is initiated directly by some user mode application. Typical examples of this use case are security checks performed by an user mode filesystem server. Responses use the same channels in the opposite direction.

4.9 Flow of events in case a security check is initiated due to a syscall invoked on a remote node. Responses use the same channels in the opposite direction.

4.10 Flow of events when a migrated process tries to access a (non-cached) file. Responses use the same channels in the opposite direction.

5.1 A graph capturing duration of an execution of a Mandelbrot set computation on a single machine with an increasing number of subtasks.

5.2 A graph capturing a duration of an execution of a Mandelbrot set computation on an increasing number of machines.

5.3 A graph capturing a number of cluster tasks running on each of cluster machines during distributed execution of a Mandelbrot set calculations with a simulated user activity on 2 nodes.

5.4 A graph capturing a CPU usage of each of cluster machines during distributed execution of a Mandelbrot set calculations with a simulated user activity on 2 nodes.
5.5 A graph capturing a duration of an execution of an increasing number of Mandelbrot set computations running on a full cluster. .......................... 85
5.6 A graph capturing a duration of a compilation using Clondike on an increasing number of nodes. ................................................................. 86
5.7 A graph capturing a duration of a compilation using dist-cc on an increasing number of nodes. ................................................................. 87
5.8 A graph capturing a duration of a compilation of 2 kernels simultaneously on an increasing number of nodes. .................................................. 88
5.9 A graph capturing a duration of an increasing number of simultaneous kernel compilations on a full cluster. ...................................................... 89
5.10 A graph capturing a measured duration of a compilation on N nodes with 5 nodes performing a simulated user activity. ................................. 90
5.11 Graph capturing a real execution time and an ideal execution time of a kernel compilation started from the node Alfa. ................................. 92
5.12 Graph capturing a real execution time and an ideal execution time of a kernel compilation started from the node Delta. ........................................ 93
5.13 Chart capturing an execution of 4 parallel kernels compilation. For each node we show a time how long would it compile a kernel on its own, and how long did it take when nodes were clustered together and each of them started its own calculation. ...................................................... 94
6.1 First, a node B gets some file X from a node A. Afterwards, when a node C needs the file X, it could get it again from the node A, but also from the node B. Getting from the node B could be advantageous if it is closer to the node C, or if the node A is getting overloaded. ................................. 97
List of Tables

5.1 Parameters of the test machines. Memory is in Gigabytes, build time in format minutes:seconds. ............................................................. 91
5.2 An average difference of predicted values from real values in percents.  . . 94
Chapter 1

Introduction

1.1 Motivation

The goal of our work was to create a clustering system, that could be safely used in various real world scenarios, where current systems fail, mainly due to their lack of security features. In this section we describe various examples of such environments and the reasons why current system are not a good solution for these environments. The examples do not represent an exhaustive list of possible uses, but rather serve as motivation use cases, to understand context of our work and the problems we are trying to solve.

Similarly as most of the other clustering solutions, the first environment considered for our system is university computer networks. Traditionally, universities were building dedicated clustering solutions [73, 50]. Those solutions serve well educational purposes, but they are not cost efficient, as they require machines to be exclusively owned by the cluster. Such machines cannot be used for any other purposes, namely they cannot be used as workstations for students. Many systems tried to address this problem by building clusters directly out of workstations. There are some well known clustering system trying to build an illusion of a supercomputer build out of workstations [62, 9], and also some simpler batch-oriented solutions [23]. A common problem with all existing solutions is the lack of security mechanisms.\(^1\) The rationale of ignoring security aspects in university computer network clusters is based on the assumption that all computers in network are fully administered by the university and nobody can gain superuser access to those computers. This assumption has several important consequences:

1. If any single machine in a cluster gets compromised, all nodes participating in a cluster are compromised.

2. The network used by a cluster must be secured from the rest of the network. Misconfiguration of the network or a breach of networking rules compromises all cluster nodes.

\(^1\)Batching systems are better in this aspect, but they provide limited functionality compared to SSI clusters. In addition, the existing batch-oriented systems do not address well security in case of process migration.
3. No external machine can be allowed to connect to the network used by the cluster, otherwise the previous condition gets violated.

4. No user can be granted superuser privileges on any workstation participating in cluster.

Point 1 represents a serious security threat that existing solutions decided to accept. Even if this may be in theory accepted in a single university context, it is unlikely to be an acceptable risk for use in industry. Point 2 is similarly a severe risk. The existing solutions do not assume only single owner, but they often assume trusted network as well. This means that a technically skilled user can send a forged message from one cluster node to another one pretending to be an original cluster message. Standard socket operations are not restricted to superusers, so any ordinary user can do this and, as a result, gain control over the cluster. A simple solution of this problem is to allow messages only from privileged ports, i.e., to allow only privileged users send such messages. However, even with the limitation of ports, no external machines can be allowed to connect to the network, as they could be owned by other users and hence they can be used to send forged messages from privileged ports. Similarly to point 1, a security breach of one cluster machine can compromise all cluster machines.

Point 3 was not a problem some time ago, but nowadays all students own their personal computers, laptops, netbooks, or smartphones. To protect cluster networks, universities have to enforce strict separation of a cluster network from the network where users connect their personal devices. However, with clusters built out of workstations, this is really an unfortunate situation. Current portable computers have a considerable computing power and in a cluster built out of workstations, this computing power could be shared if the network is shared. Similarly, the portable computing devices can benefit from computing power of a cluster system.

Point 4 is not a problem in many contexts, but in computer science schools, it may limit machines that could participate in the cluster. It is common to have laboratories where users have a full control over their machines for educational purposes, like operating system or system administration lectures. Those computers cannot be part of clusters without security systems in place.

Organizations often have many independent subnetworks that could build clusters of their own. Naturally, there is an interest to make those clusters work together, an approach typically called multi-clusters \[^2\]. There seems to be a common agreement that those systems cannot be used without any security mechanisms. However, the mechanisms proposed for those systems mainly focus on secure interconnection and coarse grain permission administration. Those mechanisms can address the most common threats, but they fail to address the risk specific to process migrations and often they ignore a subtle,

\[^2\]It is possible to perform required administrative exercises in virtual machines. However, assuming these virtual machines have access to the network, they represent the same risk as access to physical machine.
but important risks associated with the low trust, like forgery of information, failure to obey agreed cooperation protocols, or pure free rider scheduling problems [51].

An alternative to using multi-clusters are grid systems. The grid systems often have strong security mechanisms and therefore they address all security concerns. The main reason why grid-based systems are not always used in multi-cluster environments is an increased technical complexity of the overall system, as it would require some form of integration of a clustering system with the grid system. It is often quite complex to configure the grid system itself and an integration with another incompatible solution does further increase the administrative burden.

Grid systems could be, and often are, used without any clustering solution. They simply have their own specialized machines that control a subset of a network. This solution avoids the problem of complex system integration, but it is generally much less flexible than what clusters could provide especially in terms of process migration capabilities. In addition, grid systems usually require manual approval or invitation of new members, by granting them access by some existing grid member. This may be beneficial for security, but in environment where each of possibly thousands of nodes could be owned by a different owner, this restriction does not provide much additional security to the system and can only slowdown its adoption.

The clustering systems are not limited to universities. Commercial companies often build cluster systems for internal needs like compilation farms, rendering farms, or clusters running complex simulations. Those clusters can be build out of dedicated machines, but a similar argument of low utilization of high-performance workstations holds here. Commercial companies are often more sensitive to security threats than universities and usage of clustering solutions with their current security model would be prohibitive for them. In addition, in IT companies a large number of developers have superuser access to their machines and often they even connect their own machines to the network. In such a model, it would be very risky to deploy current clustering solutions.

1.2 Vision

In the context of use cases and environments for cluster and grid systems from the previous section, we now present our vision of an ideal system addressing those cases. The short-term vision is to create a system that would allow building clusters of workstations in an easy and secure way. It should not require deep understanding of computers and complex configuration like grid systems do, but it should not be open to all sorts of security attacks like current workstation clusters are. With strong security mechanisms and a secure default configuration, the system should not by default put any barriers on joining of new members, anybody should be free to join the system without any explicit approval process\(^3\). The system should be tolerant to individual node failures or unannounced departures, but it is not an objective to build system for building fault-tolerant high-available service.

\(^3\)But individual node owner should still have an option to opt out of this mechanism and disallow contact with unknown nodes.
Network partitions can affect running jobs, but they should not affect the cluster itself. The individual disconnected parts of the cluster should still work as (smaller) clusters, just without interaction with machines from disconnected parts. The system should scale at least as well as other existing systems. In addition, it should not suffer from any performance bottleneck, rather all services should be fully decentralized to allow near linear scaling of overall cluster computing power with the cluster size. The system, and especially scheduling subsystems, should not depend on trustworthiness of exchanged information, neither assume other nodes cooperate in agreed protocols.

A long term vision is to build a system, that makes use of security and scheduling mechanisms from the smaller scale cluster, but it scales to large scale environments where multi-cluster and grid systems are currently used. In contrast to multi-clusters, our envisioned system would enforce security and scheduling rules on a per node basis, rather than on per cluster basis. In contrast to the grid solution, uniform security rules would be enforced across all system down to a node level, with no specific integration requirements. In addition, the system would allow all typical services that clusters do, but in much larger scale.

Eventually, the system could be used even in large scale client server computation systems, like SETI [78] or BOINIC [8] systems. Those systems work very well, but they are very specialized solutions and require users to install a specialized system that cannot be used for any other purpose. If the installed system would provide users other benefits, like local clustering capabilities, it could be more attractive for users to use and attract larger user base.

In this thesis, we focus primarily on building a core of the system in the form of a local cluster, but all important architectural decisions are driven by a long term vision, to make further extensions easy to implement.

1.3 Contributions of the Thesis

The key contributions of this thesis are:

1. A proposal and implementation of a secure peer to peer clustering solution, that could be used in real world environments without non realistic assumptions about environment trustworthiness.

2. A security system inspired by trust management systems, tailored for usage in peer to peer clusters, was proposed and its core was implemented for the Clondike system. A similar system could be used for any other peer to peer clustering solution.

3. A flexible scheduling framework, that is extremely easy for experiments and even non expert users can create their own customized scheduling strategies.
1.4 Authorship of the Work

This doctoral thesis is a continuation of preceding work of several students from Czech Technical University. It may be difficult to distinguish some contributions of the doctoral thesis author from contributions of other people, so here we clearly separate individual contributions.

Martin Kačer PhD. is the original proposer of the non dedicated clustering system and he started the Clondike project. He investigated all other clustering solutions at that time, designed the system architecture and lead other students working on the system. Most of the system was implemented by Jan Čapek. His main contribution was proposal and implementation of migration mechanisms used in the system. In addition, Daniel Langr and Jan Kopřiva contributed to implementation of basic functionality of the original Clondike system. Michael Koštál investigated a market inspired scheduling strategies in context of Clondike system, but his work never get integrated into the system. Petr Malát contributed significantly to distributed system call forwarding infrastructure, basic IPC support, and he also authored pipe migration mechanisms.

When the author of this doctoral thesis, Martin Šťava, started working on the system, the original contributors have already left or were leaving the project. At the time of hand over the system state was as follows:

1. The system was a traditional non dedicated cluster and supported only a single core node. The core node was a single point of failure and a potential performance bottleneck.

2. The system supported with some problems preemptive process migration, the non preemptive process migration code was not present.

3. No real integration with any distributed system was in place. The last attempt to integrate a distributed filesystem was unsuccessful, as the Coda filesystem turned out to be very inefficient, see [52].

4. The system was well designed for separation of responsibilities between kernel and user mode. However, except of a few basic migration scripts no real cluster code in user mode existed. In addition, no good communication channel between user mode and kernel mode was in place.

5. The system was designed with security in mind and several security mechanisms were proposed, but apart from basic process isolation no security mechanisms were implemented.

6. No scheduler and no scheduling algorithms were present.

7. A lot of unnecessary limitations on process migrations were present, since many system call forwarding mechanisms were missing.
8. System supported 32 bit operating systems only, while most of today’s machines and operating systems are 64 bit.

Despite some existing problems and limitations the system was a very good starting point thanks to its well thoughtout design. Apart from the key high level contribution listed in section 1.3, Martin Štava contributed to the system in the following ways:

1. Proposed the idea of extending a non dedicated system into a peer to peer system and implemented this extension to the Clondike system.

2. Proposed a mechanism, how IPSec and OpenVPN could be automatically used for channel security without any direct user configuration.

3. Extended the stigmata mechanism from filesystem only protection mechanism to a generic protection mechanism to be used for any restricted access. In addition, a mechanism how could stigmata co-exist with caches was proposed.

4. Introduced a new communication channel between kernel and user mode that support communication initiation from both sides.

5. Implemented a user mode application logic, including membership management, security handling and scheduling.

6. Investigated integration with Plan 9 distributed filesystem and proposed a security mechanisms necessary for integration into Clondike.

7. Implemented several key missing distributed system calls, including the most important calls like \texttt{fork} and \texttt{wait}.

8. Implemented non preemptive migration support.

9. Implemented support for vsyscalls (through VSDO).

10. Proposed and implemented a simple file system agnostic caching layer, that could be mounted over any distributed filesystems and extend it by basic caching capabilities.

11. Support for 64 bit machines and partial interoperability with 32 bit machines.

12. Designed and implemented a framework to perform distributed measurements on Clondike clusters.

13. Proposed and implemented basic mechanism performing execution time predictions.

14. Proposed mechanism for generating distributed execution logs (implemented by Petr Malát).

Josef Gattermayer and Petr Malát also helped with performing of final system measurements in computer laboratory.
1.5 Scope of the Work

There is a large variety of options how distributed systems can be build. To make the doctoral thesis more focused, we will narrow the scope of the doctoral thesis primarily to clusters providing single system image at the operating system level. This is the most interesting type of distributed systems build out of networks of workstations and it is the main focus of our research. To be complete, we compare our solution to other distributed system architectures in Section 2.2.

The single system image clusters are traditionally implemented by modifications applied to existing operating systems, because they need to provide transparent process migration functionality. Our system also relies on this approach. Because of its openness, we have chosen the Linux kernel as a basis of our system. Therefore, we compare our systems mostly with other Linux/Unix-based systems. Nevertheless, most of our concepts are system agnostic and can be applied to any operating system.

For the environment, we assume typical x86 or AMD64-based personal computers, interconnected with traditional switched IP networks, using Ethernet protocol for local area network data transfer.

1.6 Structure of the Thesis

The thesis is organized into chapters as follows:

1. *Introduction*: Describes the motivation behind our efforts together with our goals. There is also a list of contributions of this doctoral thesis.

2. *Background and State-of-the-Art*: Introduces the reader to the necessary background and surveys the current state-of-the-art.


4. *Implementation*: Illustrates how the system could be implemented. Implementation was performed as an extension of the Clondike clustering system.

5. *Experimental Evaluation*: Various measurements on a workstation cluster were performed and results are discussed.

6. *Conclusions*: Summarizes the results of our research, suggests possible topics for further research, and concludes the thesis.
Chapter 2

Background and State-of-the-Art

2.1 Theoretical Background

Our work builds on research from 2 major areas of computer science - Operating systems and Distributed computing. In this section we introduce the key terms and concepts from those areas and point to recommended literature containing further details about the topics.

2.1.1 Operating systems

Operating systems are an important software abstraction layer, isolating user applications from hardware specifics of individual machines. They expose an uniform API for programmers of user applications. These APIs are used to interact with the system, no matter what hardware is used\footnote{Some architectural differences naturally affect programs, but the dependencies on hardware are nevertheless significantly reduced.}. Although it is technically possible to use a computer without an operating system, these days basically all computers have at least one operating system installed.

There are many well known and less well known operating system implementations. The most commonly used operating system on personal computers are currently commercial systems from Windows family produced by the Microsoft company. Another well known operating system, used predominantly on server machines, but also increasingly on mobile (smart) phones and desktops is the Linux operating system. In our work, we focus primarily on the Linux operating system as its source code is available and custom changes to the system are possible and encouraged (in contrast to closed source commercial products). However, the principles of operating systems are similar and analogous techniques could be used for other operating systems. In this section we further discuss generic concepts and we get to Linux specifics in the following section.
2.1.1.1 Programs

Terms as a program, process, application, or a job are commonly used in computer science texts, but their meaning is sometimes a bit blurred without clear boundaries between them. We explicitly define those terms here to make further discussions precise.

**Program** A code, in form of sequence of instructions, that could be interpreted and executed by a processor. Often accompanied by other non functional data, like configuration files, images, or documentation.

**Application** This term is used when talking about a program or a set of programs capable of performing some end user facing functionality. All software applications are composed of programs and the difference between program and application is mainly only a point of view so those terms are often used interchangeably.

**Job** Is an assignment of work by a user to a computer. Jobs are composed of one or more tasks.

**Task** Is a program or a part of a program being executed on a computer.

**Process** Is a task representation in the operating system. It contains all information representing a current state of task. The terms process and task are sometimes used interchangeably, but we distinguish those terms here such that task is an abstract term used for a logical piece of work, while process is a concrete entity existing in the operating system.

**Thread** We use this term for a part of process, that has its own execution status represented by a stack and instruction pointer, but shares rest of its state with other threads inside a process. A process always has at least one thread, but it can have more threads.

2.1.1.2 Operating modes

For security and robustness reasons, a majority of toady’s processors support association of execution context (i.e. thread) with a privilege level. This is usually done through a distinct operating modes of CPU, so called protection rings. Processes in lower rings have more access to a physical system resources than processes in higher rings. Typically, there is a distinguished lowest ring 0 that has direct access to hardware, while the higher rings could have more limited privileges.

Operating systems build on the protection rings mechanism and separate a code being executed into multiple security levels. Traditionally, two basic security levels are used - a kernel mode and a user mode. The kernel mode runs in the lowest privilege ring and hence has direct access to hardware. This is where the core of operating system resides. A code running in a kernel mode is simply called kernel and it forms the basis of the

---

2Confusingly enough, processes are represented by structures called task in a Linux kernel.
operating system. All user applications run in a user mode, typically represented by the highest available ring. In this mode, processes do not have access to most of the physical resources of computer, but rather have to access them indirectly by talking to the kernel.

A code running in a higher protection ring cannot access code running in a lower protection ring directly. Instead, it must use predefined methods of accessing of this code, using special gates provided by the CPU. In operating systems terminology, those gates are called system calls and they are the only way how could programs running in user mode communicate with the kernel. This protection mechanisms helps to achieve higher security and stability of the system. Because the interaction with the lower rings is limited to well defined access points, limited in number (typically at most a few hundred calls), it is possible to pay special attention to securing those calls and making sure they cannot perform any malicious action to either crash the system or gain access to some restricted resources.

2.1.1.3 Virtualization

Virtualization in computer science refers to a process of presenting some (virtual) resource, that do not exist in reality, to the resource consumer. The most interesting form of virtualization related to our research is hardware virtualization, where a part of physical computer is presented to the operating system as if it was a real computer of that properties. This way, the computer could be divided into multiple virtual computers, each of them running an independent and possibly different operating system. Such a virtualization is useful both in workstation computers, where users can run multiple version of operating system on one computer, but also on the servers, where resources could be shared between multiple independent applications in a more economical and secure way.

The virtualized environment is provided through virtual machines. Those virtual machines have similarly as regular machines memory limits, bounded count of CPUs and its frequency, and some peripheral devices. Unlike as with the regular machines, the resources of virtual machines do not exist exactly in the form as presented by the virtual machine, but they are typically larger. For example, a virtual machine can be configured to have 1GB of RAM and 1 CPU running on 1Ghz, while the physical machine running this virtual machine can have 8GB of RAM and 4 CPUs each running on 2Ghz frequency. A single physical machine could host multiple virtual machines.

An operating system running inside a virtual machine is called guest operating system. On workstation computers, the virtual machine is usually run directly from the main operating system on that machine, in this case the operating system from which is the virtual machine executed is called host operating system. On servers, there is usually a special layer installed directly on the physical machine called hypervisor. The hypervisor hosts directly other virtual machines.

The hardware virtualization used to introduce a non negligible performance penalty on operating systems and applications inside a virtual machine. However, recent introduction of special hardware support for virtualization reduced those overheads and typical slowdown on those machines compared to real machines of equal parameters do not exceed 10
percent.

2.1.2 Linux OS

In this section we briefly describe key components and subsystems specific to the Linux operating system. We only talk about those part that are necessary to understand topics discussed in this thesis; a full description is way beyond a scope of this work. A good literature about the topic is [18] or [57].

2.1.2.1 Memory management

Handling of memory in a shared environment, where many processes and users compete for limited resource in form of available RAM, is a challenging task. It has a significant impact on overall system performance and so a large part of kernel complexity resides in memory handling. It is not our ambition to describe this subsystem in much detail, there are books devoted just to this topic, for example [35].

The most important aspect of the memory subsystem for our work is memory virtualization. All processes running in user mode access memory through virtualized addresses. Historically, there are 2 main mechanisms for memory virtualization - memory segmentation and memory paging. Segmentation allows (virtual) division of memory into smaller logical blocks of variable length with different access privileges and possibly different offset in a physical memory. A newer concept, memory paging divides memory into small blocks called memory pages, typically 4KB large. Both of those systems are for performance reasons supported directly by hardware. Linux uses segmentation only for a rough memory access control and relies mostly on memory paging for actual memory management.

The paging allows mapping of virtual memory pages to actual physical memory. The physical memory is divided into page frames and each logical memory page could be mapped through a virtual address to some page frame. The mapping of logical pages to page frames is not one to one, it is possible that multiple logical pages point to a same page frame. Some logical page may not actually point to any page frame at all (it either has not been initialized yet, or it has been swapped to disk). Paging is the mechanism that allows running concurrently programs using same logical memory addresses without interfering, as each of them has its own mapping of logical addresses to page frames. It also serves as a protection mechanism, because processes can access only pages they have mapped and mapping is controlled by kernel.

In order to save memory on demand paging is used. On demand paging means, that assignment of a logical address to a process and actual mapping to physical memory are separated. First, a process asks kernel for some logical address range to be allocated to his address space, i.e. a set of allocated memory ranges. After a successful allocation, the memory still does not need to be mapped to a physical memory and its mapping happens only when the memory page is referenced for a first time. If the page is not mapped at the time it is referenced, so called page fault handler is invoked to resolve the mapping.
2.1. THEORETICAL BACKGROUND

The page fault handler is a code residing in kernel that performs mapping between the logical addresses and physical page frames.

Our system exploits paging mechanism during process migration (see Section 4.4) and custom caching layer (see Section 4.10).

2.1.2.2 System calls

As mentioned earlier, system calls are the way how programs running in a user mode interact with the kernel. On older machines\textsuperscript{3}, the system calls were implemented using CPU interrupts, namely interrupt 0x80 on Linux. A calling process fills a specific call code and its arguments into predefined processor registers and invokes the interrupt. An interrupt handler in kernel checks arguments and delegates the call to specific system call handler. Finally, the control is returned to a user mode, passing result back in predefined registers.

Modern CPUs contain special instructions for performing system calls that are generally much faster than older interrupts. Since the instruction is not available in all CPUs and neither programs nor operating systems are normally compiled for a specific CPU, the instruction call cannot be directly hardwired into the operating system. Instead, Linux uses a special trick based on a virtual dynamic shared object, often referred as vSDO. The vSDO is a virtual memory page, that is mapped to a (random) location of every running user mode process. A content of this page is dynamically generated on system start and it detects whether special system call instruction is supported and if so it generates code relying on this instruction, otherwise old interrupt style invocation is used. Programs use this page to invoke system calls, so they do not need to care whether the instruction is supported and the most efficient system call execution technique is automatically used.

2.1.2.3 Processes and threads

As per our definition a process is a representation of a program being executed. Technically, the kernel itself could be considered a process, but it is usually not referred as a process. Each process consists of one or more threads. There are 2 main types of threads - green threads and native threads. Green threads are completely invisible to operating system and they are fully controlled and scheduled by the user mode program itself. They incur less overhead, but do not allow real concurrent execution. A more interesting type of threads for our work are native threads that are managed by the kernel. In Linux, each native thread has a corresponding structure in the kernel, called a kernel thread. A process is represented in kernel by a collection of kernel threads corresponding to all its native threads. Not all kernel threads have corresponding user mode processes, some of them exist only in kernel where they perform some support functionality, for example swapping.

In addition to a kernel thread, each native thread has an in memory kernel structure describing its state. The structure keeps track of memory allocation, open file, pending signals, scheduling information, etc. Since there is one such a structure per each native user

\textsuperscript{3}Roughly 6 years ago.
mode thread, it means user mode processes could have multiple corresponding structures in kernel. The structures for individual threads belonging to a single process are independent, but they have some common properties, for example they share access privileges, open files, and use same memory mapping and signal handling structures.

Each thread in the system has assigned unique identifier called thread identifier (tid) and each process has a unique identifier called process identifier (pid). All threads belonging to a same process share same process identifier. Both process and thread identifiers are assigned by the kernel and they are visible to user mode programs.

At the time, there could be only one thread running on a particular CPU\(^4\). There is often more threads than CPUs that want to run at some time and so operating system has to share CPU time between them. A process of changing a running thread on CPU is called context switch. It involves updating all processor registers to reflect state of a new thread and also updates of in memory structures pointing to current process state. The most costly part of context switch is normally a switch of memory mappings as it requires invalidation of current memory mapping caches. The memory invalidation does not happen if the system is switching between 2 threads sharing same memory mappings, i.e. 2 threads from a same process.

The context switch happens when a thread switches to a kernel mode to execute some privileged action. It could be either a system call, signal handling or a fault handling. Some processes may not invoke such an action for long time, so there are artificial time interrupts generated that allow context switches at predefined intervals, called time quanta.

When the system starts, it creates only a few basic kernel threads that are required for operating system to work. All other threads are created using a fork system call to make a copy of some existing thread. This system call creates a copy of a thread that invokes the call and establishes child - parent relation between those two threads. The new thread gets its new identifier assigned, it gets its own stack and depending on parameters of the system call it may also get a new address space assigned. In case the thread should be part of a same process as the thread being forked, new address space is not assigned and instead the old one is shared. Even if the address space is not shared, the new process gets exactly the same copy of memory as the original process, though in practice some optimization techniques are used to avoid need of unnecessary copying of memory.

Often, after the fork system call is followed by the exec system call. This system call erases memory allocated by current process and loads a new binary into memory and starts execution of the code from a loaded binary. The child - parent relation is not affected by the exec call.

### 2.1.2.4 Filesystems

Filesystems in operating system serve as an abstraction layer between a publicly exposed operating system interface to files and a physical storage of data on disk. The access to filesystem is exposed through a set of well known system calls, like open, read, or close.

\(^4\)Or more precisely one per CPU core and even more in case of CPUs with hyperthreading support.
In addition to standard operations, Linux also supports memory mapping of files directly into a process memory. After that, a process could access the mapped file the same way as it does access other parts of its own mapped memory.

All associated system calls and access operations are delegated to a generic virtual filesystem (vfs) interface. There are many different implementations of filesystems available in Linux. Each of the specific filesystem implementations uses its own internal data organization optimized for various goals, but they all expose this functionality through unified vfs interface. The Linux filesystem is not limited to files stored physically on disk, but the same vfs interface is used also for network filesystems, where the files exist on remote computers, and as well for pseudo filesystems, where the files do not exist at all, but access calls are directly interpreted by the kernel. The pseudo filesystems are often used for exposing some interesting data from kernel, or as an interface to control certain kernel parameters.

Linux files are organized in a hierarchical structure, having a unique root directory identified by a special path '/'. All other files and directories are either direct or indirect children of the root directory. In order to make filesystem accessible to user programs, it has to be mounted somewhere. Filesystems are mounted to existing directories, except of the root filesystem that is mounted first and contains the root directory. When a filesystem is mounted to a directory, it becomes transparently accessible to other processes at the path it was mounted. If there were any files and directories at the directory where the filesystem was mounted, they will be hidden by this filesystem until it is unmounted.

Traditionally, all processes shared the same view of filesystem, but newer versions of Linux support private namespaces, where a filesystem could be mounted only in a namespace of a certain process, but other process do not know about this mount and see only the original directory. In addition, each process (with sufficient privileges) can change its root directory by invoking a chroot system call. After invoking this call, process namespace could be changed to some subdirectory of the original root directory. All references to files after this call will be relative to this new root directory and files residing outside of a file tree rooted in a new root directory become inaccessible to the process. All threads within a single process share the same filesystem namespace and root directory.

When a process communicates with files using the system calls, the kernel needs to keep track of files open by this process, current read positions, or flags used when a file was open. The kernel keeps in memory structures per each open file to track those data. The kernel structure is opaque to user mode processes, but they can refer to it by a special handle (file descriptor) they get when they open a file. The structures representing file state are shared by all threads of a same process.

2.1.2.5 Pipes

Linux heavily uses context of pipes. Pipes are simple data queues, where one or more process can write data and other processes could read these data. The data are kept only in kernel memory and they are not stored anywhere. There are 2 types of pipes - standard pipes and fifos. The only difference between these two is, that fifos are named and can
be accessed by any process with sufficient privileges, while pipes are not publicly visible and exist only between a pair of related (child-parent) processes. Pipes are used in most of the interactive communication between the system and user for dispatching of output data. Also any command chaining between the system and user for dispatching of output data via the pipe character results in pipes being used. Pipes are exposed to system using the uniform vfs interface, so they are conceptually very similar to pseudo files.

2.1.2.6 Executables

Each program needs to have at least one executable file. This file contains code to be executed by the system and some other metadata about the program. There are many formats of those file, one of the most commonly used is the elf binary format. When a file is being executed, it is passed to a list of registered binary file handler for execution. The handlers are tested one by one, if they can execute the specified binary format. The first handler capable of executing this format is used.

It is possible to register a custom binary handler and we use this functionality for restarting of checkpoints as described in Section 4.4.

2.1.2.7 Signals

Signals were introduced as a simple inter process communication mechanism. A standard signal is a message with no arguments sent by some process to another process. Each standard signal corresponds to a single predefined number, for example a well known SIGKILL signal, used to kill a process, corresponds to code 9. There are 31 standard signals, but some of them are user defined and their behavior is unspecified. We exploit the undefined signals in our process migration mechanism as described in Section 4.4.

A signal is sent to a process using a system call. When a process receives a signal, a corresponding signal number is marked as delivered and the process is forced to handle the signal. If multiple signals of the same type arrive before the process handles the signal, the further repeated occurrences of the same signal are ignored. Handling of a signal is initiated from the kernel mode, but each process can install its own custom signal handlers that reside in the process user mode code.

In addition to standard signals, Linux support also an extension called realtime signals. Those signals do not have kernel defined meaning and in contrast to standard signals repeated occurrences of the same signal are queued and delivered in the same order they were generated.

2.1.2.8 Networking

Linux supports most of currently used networking technologies. Communication is abstracted through a socket interface. A socket represents communication endpoint and it is a way user programs interact with the network subsystem. First, a user program has to create a socket using a system call. The system call returns a descriptor of that socket and this descriptor is used as a parameter to following system calls performing reads and
writes on the socket. While the user mode programs have to use system calls to interact with sockets, it is possible to open sockets directly inside kernel and thus establish remote connections directly from the kernel.

Apart from standard networking protocol, Linux kernel supports also some custom internal protocols used only for local communication within components of one computer. An important example of this protocol is Netlink protocol, that is designed to be used for structured communication between kernel and user mode code.

2.1.2.9 Access control

Linux systems traditionally distinguished a special root user and other users. The root user can do any operation with the system, while other users can perform only a subset of possible operations. Newer versions of Linux systems are based on capabilities. Capabilities allow more fine grained assignment of permissions, so each user could have associated a different set of operations he can perform.

Each user has its unique identifier called user identifier (uid). Root user traditionally has uid 0. Users could be assigned to one or more groups. Each group has its unique group identifier (gid).

Every running process has associated uid and gid. Those are determined at process creation or a binary execution and they could be as well changed by a sufficiently privileged user by using the chown system call. Process uid and gid determines what (privileged) actions can the process perform.

A special case of access control is applied at the filesystem layer. Each file has exactly one owner and exactly one owning user group. For every file there are 3 sets of permissions - one for owner, one for owning group, and one for all others. Each of those sets contains any combination of 3 possible privileges - read, write, execute. When checking if a process P can perform an operation O on a file F, the following logic is used:

1. If the file F allows the operation O to any user, then the operation is allowed.
2. If a uid associated with the process P refers to the owner of the file F, and the owner is allowed to perform the operation O, then the operation is allowed.
3. If a gid associated with the process P refers to the owning group of the file F, and the owning group is allowed to perform the operation O, then the operation is allowed.
4. Otherwise, the operation O is denied.

There is also a newer more flexible access control mechanism, that allows users to specify access permissions to a file to any user or group, not just to owners. The mechanism is called access control lists (ACL). ACLs are not so commonly used and some filesystems do not support them at all.

\footnote{In reality, the situation is more complex and process has multiple pairs of uid and gid, but only one effective at the moment.}
2.1.2.10 Modules

Linux supports dynamic loading and unloading of parts of the kernel via the modules mechanism. A module is a binary file containing a piece of code and data. When loaded to the kernel, the functionality provided by the module could be used by other parts of the kernel. Modules could have dependencies between each other, i.e. a module A may be using some functions provided by a module B, so when the module A is being loaded the module B has to be loaded as well. A large part of the Linux kernel could be compiled as modules. Typical examples of modules are various filesystems or network protocol implementations.

The modules have 2 main advantages. First, code could be loaded into a memory only when it is required and unloaded when it is not required any more, thus saving a memory. The second advantage is in a cleaner separation of the code, since a module consists of a self contained piece of functionality, so the resulting code is well structured with clear dependencies. We exploit modules for most of the kernel mode functionality of our system.

2.1.3 Distributed computing

Distributed computing is an old well established discipline of computer science. The distributed computing research focus is in distributed systems. Distributed system is a system formed by multiple computers interconnected via a computer network. The computers in a distributed system have to run some common protocols in order to be able to communicate with each other and they typically cooperate on some common task.

In this section we introduce the basic terms used throughout this doctoral thesis. For more details about distributed computing we recommend for example [81, 24].

2.1.3.1 Computers and nodes

Each distributed system is formed by at least 2 computers and a network interconnecting those computers. In distributed computing terminology, computers are called nodes. Each computer has one or more central processing units (CPU) that are used for execution of code. In case a computer has multiple CPUs, the CPUs are typically interconnected through an internal bus and through this bus they share access to common computer components like memory or IO devices. Recently, due to hardware problems with increasing operating frequency of CPUs, manufactures started introducing multi-core CPUs. A multi-core CPU has a multiple (mostly) independent execution units and so it is capable of executing multiple threads concurrently. The cores of a single CPU typically share some CPU caches and they compete with each other for access to a shared bus. It is possible to build computers with multiple multi-core CPUs.

Computers differ by their purpose. Computer build to be used directly by users are called workstations. Those computers come equipped with screen and all peripheral devices like keyboard and mouse. Current computers typically contain a single multi-core CPU, low end computers having typically just 2 cores, while high end computers can have 6 or more cores. Servers are computers build to serve remote requests and they typically
do not have any peripheral devices or screens. They are always connected to some network as there is no other way how to use them. From the hardware point, servers could have a similar equipment as workstations, but they are often also equipped with a specialized more reliable hardware and multiple multi-core CPUs.

2.1.3.2 Networking

Employed networking technologies usually differ depending on a context they are used. There are two major categories of networks: local area networks (LAN) and wide area networks (WAN). LAN networks are represented by computers connected within one room, or possibly within one building. WAN networks are large networks spanning multiple buildings, cities, or even countries, the largest example of WAN is the Internet. WAN networks often interconnect multiple LAN networks together.

LAN networks are predominantly build using Ethernet [1] on the data link layer (layer 2) of the OSI model [20]. Each computer has a unique data link address, in case of Ethernet this it is called MAC address. Computers at the data link layer could be either directly connected using a cable, or more commonly they can be connected by a cable to a network switch. Switches operate at the data link layer and they are capable of interconnecting multiple computers at this layer. The simplest network topology formed by a switch is so called star network. In this topology, all computers are connected to a single switch and they communicate with each other using this switch. In a more complex topology, multiple switches are connected together and each computer is connected to one of the switches. Switches automatically break possible loops in the network [6] and provide unique path between each pair of computers. A set of all computers that could talk to each other on a data link layer is called a network segment. Data in Ethernet networks are exchanged in small chunks of bytes called frames. A frame could be addressed to a single computer from the same network segment by using its MAC address, or to all computers in the same segment when a special broadcast address is used. Therefore, network segments are also sometimes called broadcast domains.

IP protocol suite [39], operating at the network layer (layer 3) of the OSI model, is a higher level protocol typically used over the Ethernet. Data in IP network are exchanged using IP packets, or simply packets. Each computer at the IP network has an IP address assigned. Addresses are logically grouped into subnetworks. A subnetwork is defined by its IP address and a network mask. An example of subnetwork could be a network of IP range 192.168.1.0 to 192.168.1.254 defined as IP address 192.168.1.0 and a network mask 255.255.255.0. Computers belonging to a same subnetwork must belong to a single Ethernet segment, but there could be multiple IP subnetworks on a single Ethernet segment. Devices delimiting boundaries between subnetworks are called routers. A process of sending a packet from one subnetwork to another is called routing. IP protocol could be used also on other networks than Ethernet, for example in LANs IP protocol can run over 802.11 protocol [2] used in wireless networks.

6 Using the Spanning tree protocol.
CHAPTER 2. BACKGROUND AND STATE-OF-THE-ART

WAN networks are complicated with complex topologies and protocols supporting the network infrastructure. From the user point of view, it is important that those networks also support the IP protocol and so they could be used to interconnect IP based LAN networks together. A protocols used on the data link layer vary a lot, depending on a physical medium used. A typical connection media are cables (phones, cable TVs, optical cables), but other media are used, for example microwave, satellite, point to point WiFi, etc. The details of datalink protocol are abstracted by the IP protocol, but the performance characteristics like a throughput, latency, or reliability can vary a lot.

WAN networks can be very large, the largest of the being the Internet. It turned out, that the original imposed limit on count of IP addresses is too low and it is not possible to assigned each computer on the Internet a unique IP address. This problem is addressed by a new version of the IP protocol - the IPv6 protocol. Apart from other enhacements, the protocol extends the length of addresses so that it is unlikely it could ever be exhausted. However, adoption of IPv6 is slow and it is currently not usable by everyone. A current solution to the shortage of addresses is based on a technique called network address translation (NAT). This technique allows translation of between IP addresses and it allows multiple IP addresses to be mapped under the same IP address. NAT is performed by routers or firewalls. A computer whose real IP address is same as an address seen on the Internet (or mapped 1 to 1 with such an address) is said to have a public IP address. Computers that do not have such an address cannot be reached directly by computers from the Internet, but they could themselves communicate with other computers in their local network and with computers on the Internet that have a public IP address.

Applications do not usually work directly with the IP protocol, but they rather rely on some higher level protocols. Among the best known and most commonly used protocols are the TCP and UDP protocols. The UDP protocol closely mimics underlying IP protocol as it is a connectionless, unreliable protocol with no guarantees about message delivery order. It also support broadcast within a single subnetwork. In contrast, the TCP protocol is connection oriented, i.e. 2 computers need to establish a bidirectional connection before they can talk to each other using the TCP protocol. The protocol guarantees message delivery and as well delivery order.

2.1.3.3 Goals

Distributed systems are being build with various goals. Among the most typical goals are:

**High performance** These systems are very common in research projects, as they allow solving complex problems in a reasonable time. Ideal systems offer liner speed-up of calculation with a growing number of participating computers, though in practice the speed up is usually slowing down with a size of a cluster.

---

7 Devices similar to routers, but their primary purpose is to enforce security rules between network boundaries.
2.1. THEORETICAL BACKGROUND

High availability and fault tolerance Often the goal is to build a system that is resilient to crashes of individual computers and ideally provides uninterrupted services to its users in case a limited set of computers fails.

Efficient resource utilization Observation of resource utilization on workstations have revealed, that users require full power of available resources only for a limited time, and most of the time the resources stay idle. Some systems make it possible to use idle resources of one computer by users of other computers that currently need more resources than they have. Another approach is to provide all resources as a shared resource and provision them to users as they need this. Recently, this approach is being used not only for workstation programs, but as well for server applications.

Data sharing Some distributed systems are build with the main intent of sharing of data among participating users. A typical examples of those systems are peer to peer file sharing systems like BitTorrent [22].

Resource federation Similar to data sharing, but sometimes also other types of resources than data may need to be shared, for example some rare hardware.

In our work, we primarily focus on the efficient resource utilization and high performance.

2.1.3.4 Distributed system architectures

There is a large variety among high level architectures of individual distributed systems. Following is a brief overview of the most common architectures related to our work.

Clusters Computer cluster is a set of interconnected computers cooperating on some common goal. In clusters, the computers are typically connected using a LAN network and they are all co-located in a single room or building. Traditionally, computers in a cluster were owned by a single owner and so security is often ignored in clusters.

Clusters are build with various goals, most commonly for high performance or high availability. Depending on a role of computers in a cluster we distinguish dedicated and non-dedicated clusters. In dedicated clusters, all computers become fully available to a cluster and they cannot be used as workstations at the same time. In non-dedicated clusters, some of the computers are not fully owned by the cluster and they can be used as workstations at the same time. Typically, cluster does not send any processes to non-dedicated nodes, where a local user is active at the moment.

Grids Grids are also formed by computers interconnected to perform some common goal, but unlike as in clusters, where all machines are co-located, the machines in grids are typically geographically dispersed. Grids are formed by many independent institutions cooperating together, while in clusters all computers typically share a single owner. Therefore, security is of a much higher importance in grids than in clustering systems. Apart
from security agreements, the participants usually have to agree on provided quality of service by each institution and those numbers should be honored.

The building blocks of grids could be normal computers, but grids could also be build out of mix of computers and clusters. A special case, when the system is build only from clusters is sometimes called a **multi-cluster** system. Those systems are often not considered to be a grid system as they are usually implemented only as an extension of a standard cluster and a security mechanisms are not so developed as in grid systems.

Grids are most commonly build for performance reasons and for secure data sharing reasons.

**Cloud computing** Cloud computing is a relatively new architecture, that become popular in recent years. Technically, it is still a cluster, however a main focus of this architecture is to provide strict separation between the role of the system provider and system consumer. From the provider point of view, cloud is just a cluster running one or more virtual machines on each of the computers. Each of those virtual machines belong to one of the consumers and those virtual machines could be on demand provisioned or unprovisioned. The provider of the service must ensure efficient distribution of those virtual machines on physical machines and take care of all the infrastructure. The consumer of the service just sends request to provision or unprovision his virtual machines and then only interacts with those machines, without taking care of the underlying computers or network.

Cloud computing systems are build for efficient resource usage reasons. The providers take advantage of economy of scale, where they provide the service to a large number of clients on a same physical infrastructure. The consumers benefit from the fact they do not need to take care of physical hardware and they can easily scale their resources up and down as they need.

**Peer to peer systems** The name peer to peer system comes from the fact, that each node participating in such a system is functionally equal to the others, there is no traditional distinction between clients and servers. The main benefits of peer to peer systems is a high tolerance to node failures and high scalability to millions of nodes. Those benefits are enabled by high autonomy of individual nodes and carefully designed algorithms, that always work only with a local system state knowledge.

Peer to peer systems become initially popular as a file sharing networks, but recently they are also being used for computing resource sharing. Ideas of peer to peer systems are also being integrated into some grid systems [20].

### 2.1.3.5 Single system image

**Single system image** (SSI) clusters are a special subtype of clustering systems, that virtualize a cluster and presents it to users as a single powerful machine. There are various degrees of virtualization, some systems have just common file system space, but the most
advanced systems do virtualize most of the resources and so all machines in a cluster share process space, input/output, IPC, or even an (external) IP address.

A motivation for building of such systems is to make administration and working with clusters as easy as possible. The idea is, that users are used to work with standard computers and so the interface provided by the cluster should be as similar to a standard computer as possible. It is, however, not possible to achieve full transparency, since there are some fundamental differences that do not exist in a single computer environment; among the most noticeable from user perspective is non uniform performance characteristics and unpredictable failures of hardware or network.

2.1.3.6 Data sharing

Any application that has to benefit from parallelization speedup needs to support concurrent execution of its parts. These parts are executed either as separate threads of a single process, or as multiple distinct processes. In either case, they normally need to communicate with each other in order to share data and coordinate the overall execution. There are 2 basic approaches for data sharing - message passing and shared memory. We briefly discuss both approaches and how do they transfer into a distributed systems environment.

**Message passing** Message passing is a technique where data are being exchanged by sending messages. An advantage of this technique is in its clarity; it is always clear when data are being exchanged and there is no high risk of inadvertently modifying some shared data. The technique could be used both for communication between threads and processes running on a single computer, and as well to communicate between process running on different computers.

A notable example of message passing technique is the Message passing interface (MPI) standard [29], that is commonly used for implementing of parallel computations. There are multiple implementations of this protocol (for example OpenMPI [67] or MPICH2 [63]) and they could be used both in a small systems and as well in the largest clusters build up to date [7].

The message passing technique is predominantly used in distributed systems, but thanks to its advantages in code clarity, it is increasingly more being used in parallel systems running on a single computer [27, 34].

**Shared memory** Shared memory technique enables exchanging of data as if the read/write operations were standard memory access operations. The basic idea behind this technique is to enable programmers to use the same programing style they used in non parallel programs for writing of parallel programs. Another advantage of the technique could be higher efficiency in resource utilization as multiple execution units can directly shared data without sending an access request. However, because the access to shared data looks in a code exactly the same as access to private data, it is very hard to distinguish this type of access in code, and thus it is very error prone [55].
The technique is very often used in multi-threaded applications intended to run on a single computer. In distributed systems environments a technique called distributed shared memory could be used. A distributed shared memory is just an illusion of a real shared memory provided by the operating system. From the application point of view the behavior is same as in a case of single computer, except of the performance characteristics. Access to a remote memory is typically a few orders of magnitude slower than access to a local memory. Because memory access is not so obviously expressed in shared memory applications code, it is very easy to introduce a lot of bad memory sharing patterns and applications written without a distributed shared memory in mind would generally perform very poorly. For this reason distributed shared memory is not commonly used.

2.1.3.7 Process migration

Many clustering systems are based on idea of process migration. This technique allows execution of processes on remote computers. There are 2 basic types of process migration:

Non preemptive process migration This basic type of migration allows executing processes on remote machines, but the machine has to be chosen at the beginning of execution. Due to its nature, it is sometimes referred as remote execution. The technique can be implemented without changing the kernel mode code, but it places more burden on the scheduler, as bad decisions cannot be easily corrected.

Preemptive process migration Preemptive process migration allows moving of processes between machines at nearly any time during their execution. This is a powerful technique that is technically harder to implement, but it is more flexible as it allows system to adjust process distribution based on current conditions. It is especially important for clusters build of workstations, as processes running on idle workstations may need to be migrate away when a local user becomes active again.

Systems based on virtual machines often support also a virtual machine migration. While this technique is very powerful and supports migration of nearly any program, it is much more resource intensive than a process migration, because it requires transfer of all memory pages used by the operating system and all running processes.

2.1.3.8 User roles

In cluster systems, there are several types of users involved and each can play a different role. The main roles include:

Owner Owner is a user who owns a workstation or a (part of) networking infrastructure.

Administrator Administrator is a user with super user privileges on a computer (or network devices). Owner is typically as well an administrator, but there could be more administrators of a single computer.
2.2. Previous Results and Related Work

Our system is based on the Clondike system developed by students and researchers from the Czech Technical University. One of the lead authors of Clondike system, Martin Kačer, published extensive comparison of the original Clondike system with systems from that time (2006) in his PhD Thesis [46]. His comparison of original Clondike to other systems is still valid as none of the systems has undergone any significant architectural changes since the time of his PhD Thesis publication. The comparison was heavily focused on migration and security mechanisms and was centered around cluster systems. Here, we present an up to date comparison of our new system based on Clondike with other major systems, but we do not limit ourselves to cluster systems, but consider also grid and peer to peer systems. We also compare our new system with the original Clondike system and highlight the major differences.

2.2.1 openMosix/Mosix

Mosix [62] is one of the best known clustering systems; its start dating to year 1999. Mosix initially started as an open source research system, but later it was turned into a closed source commercial system. At that point a fork of the original code was created and thus openMosix project started. The openMosix [66] project has ended on year 2008, but a group of volunteers is trying to continue with the project under a new name LinuxPMI [56].

The goal of Mosix project is to create a clustering system build out of ordinary workstations with some SSI features. The system functionality is based on process migration support. The nodes of the system exchange data about their state through a gossip protocol and the scheduler tries to balance the task distribution based on current load of nodes. The main distinguishing feature of Mosix from other SSI clusters is that it does not provide a uniform shared filesystem across the cluster, but rather each of the nodes provides its own version of filesystem to processes started from that node.

A large part of the system is implemented directly in the Linux kernel which makes the system powerful, but harder to modify. The open source version is still not migrated to new kernel versions.

The main difference between our system and the Mosix is, that the Mosix system was since the beginning designed to be used in a trusted environment with a single owner. Therefore, it lacks security mechanisms to allow resource sharing between computers with different owners. There are some basic mechanisms supported for a Mosix multi-cluster extension, however those do not address risks associated with migration as we discuss in Section 3.4.3.
2.2.2 OpenSSI

OpenSSI [68] is an ambitious project supporting probably the largest number of SSI features. There seem to be no activity in the project since year 2010, so the system is supported only on older version of the Linux system.

OpenSSI represents a dedicated clustering solution, all computers participating in a cluster have to be fully controlled by the cluster and they cannot be used for any other purpose. The system supports both preemptive and non preemptive process migration. Among the standard supported SSI features is shared filesystem, shared user space, and shared process space. Among the less common features is a shared public IP address and a high availability support.

Compared to OpenSSI, our systems allows individual computers to be independent workstations, each owned by a different user. The workstations could be normally used by their owners and cluster can use them only when they are idle. Our system provides SSI features with respect to a single computer, while OpenSSI provides SSI features globally across all participating computers. Therefore OpenSSI can provide also high availability features, while our system is not built to provide it. Since OpenSSI is a dedicated cluster, it does not have any cluster security mechanisms as our system does.

2.2.3 Kerrighed

Kerrighed [50] is one of the most recent clusters aiming for providing of SSI features. It is conceptually similar to OpenSSI, also providing a global SSI features in a cluster build of dedicated computers. Similarly, there is a very little activity after year 2010.

The project has a similar set of features as OpenSSI, though high availability or single public IP address are not supported. On the other hand, the project used to support distributed shared memory and experimental support for migratable sockets, but those features were removed as a part of code stabilization process.

The project differs from ours in the same way as OpenSSI project. Moreover, it does not support addition and removal of nodes while the cluster is running. This is an essential feature of our cluster, as each owner of machine could decide to join or leave the cluster at any time.

2.2.4 NOW

Network of workstations [50] (NOW) projects is one of the oldest attempts to build a cluster from workstations. Its main idea was to allow workstations to be used both by interactive users and by the cluster at the same time. Distributed operating systems at the time of NOW creation were typically build from scratch, while authors of NOW came with an idea to build a system as an extension of a standard existing system (Unix). The project was finished around year 1999.

The machines in NOW are not dedicated and they can work independently as workstations. Each of the machines could in addition participate in a single shared cluster.
Most of the cluster functionality was provided by an extension layer over the Unix systems called glUnix. The system was designed to support both preemptive and nonpreemptive migration, but according to paper [71] only non preemptive migration was supported. For cluster file sharing NOW used its own novel fully distributed serverless file-system.

The main differences between NOW project and our project is in security. NOW assumed deployment in a fully trusted environment, all computers have to be owned by a single administrative entity. Another important difference is in cluster representation. In NOW, each workstation acts as an independent computer and also participates in a single shared cluster. However, in our solution each node can form its own independent cluster and no cooperation between individual clusters is assumed. The glUnix library is designed in a traditional master-worker style, so a single computer in NOW cluster has to act as a master. In our system, all computers are equal and act in a peer to peer style. Our system also supports both preemptive and nonpreemptive migration.

2.2.5 Condor

Condor [23] is a batch scheduling opportunistic clustering system. It is implemented purely in user mode and it can run with standard user privileges so it introduces only limited security risk to the system. The system is widely used and still under active development.

A single Condor cluster consists of a pool of machines. A single machine in this pool is configured as a master. This machine collects resource information from other computers and it acts as a scheduler. Other machines can serve either as execution nodes, submit nodes, or both. Condor supports non preemptive migration for unmodified applications, and preemptive migration with checkpointing for applications linked with a special Condor provided library. Condor does not provide SSI features and the participating machines retain their autonomy, i.e. they could still be used as ordinary workstations.

Thanks to its user mode implementation, Condor is probably one of the most mature clustering system of these days and it is very popular. However, the user mode implementation also brings some limitation. Not only the processes have to be linked with a special library to support migration, but many features supported by SSI clusters are not supported, for example interactive jobs, or fork/exec system calls. The job scheduler of Condor performs sophisticated decision and is able to track even job dependencies. Our scheduler is simpler, however unlike Condor scheduler it also works in environments where other parties do not cooperate. Condor does have basic security systems to protect cluster jobs and users, however it still lacks fine grained security and migration protection mechanisms that our system has. Specifically, authors of Condor explicitly warn against granting join permission to computers, where administrators are not trusted.

2.2.6 Beowulf clusters

Beowulf cluster [73] was a term often used to refer to a cluster build from commodity computers running a Unix (or similar) version of operating system. A Beowulf clusters were build from dedicated computers, each of them running identical version of operating
system. A single computer acted as a master and all others as workers. Even though lot of current clustering systems are based on Linux, they are not referred as Beowulf systems any more.

Beowulf clusters could have different implementations, but they all generally assumed a full ownership of computers and so no security features were required. Computers participating in the cluster could not be used as workstations as they can in our system. Dynamic addition and removal of nodes was typically also unsupported.

2.2.7 Globus toolkit

Globus toolkit \cite{30} is de facto standard for implementation of traditional grid systems. Rather than a system, Globus toolkit is a set of components that could be used to build a grid system. The toolkit allows building of a secure and flexible resource federation grids. The toolkit is still under active development and being actively used by many organizations around the world.

Grids based on Globus toolkit differ in many ways from our system. Although they provide strong security system, the system is not decentralized and admission to the system is strictly controlled by current members or administrators. Globus based grids do not provide preemptive process migration and they do not support transparent migration or load balancing. Instead, if an application has to be executed in the grid a special job descriptor file has to be provided and the job has to be submitted to a grid. Globus also does not provide any SSI features.

2.2.8 XtreemOS

XtreemOS \cite{85} project is a recent large scale grid project funded by European Union. The project tries to differentiate from other clusters by providing simple interface to users and trying to avoid need of very complex configuration. The project is still under active development.

Unlike other grids, XtreemOS provides both non preemptive and preemptive process migration. The grid does not aim to provide SSI functionality globally, but it is possible to use local SSI clusters \footnote{The technology for local SSI clusters is based on the Kerrighed. However, since the latest release of XtreemOS local SSI clustering is discontinued.} as building blocks of the grid. The security mechanism of XtreemOS is based on virtual organization \cite{14} concept. The system is very flexible. First, users register themselves and their resources to the system. After registration, they can create virtual organizations and join their resources to these organizations. Comparatively, our system is simpler, each machine forms its own logical virtual organization and user does not have an option to form more virtual organizations or selectively assign resources. However, our system is fully decentralized, while XtreemOS system relies on central administrative entity for granting root certificates. XtreemOS also needs a lot of services to be preconfigured and their location predefined, while our system is fully autonomous and all services on local network are automatically discovered with no configuration required.
2.2.9 OurGrid

OurGrid \cite{11} is a Java language based peer to peer Grid system. It was originally based on the JXTA \cite{82} peer to peer protocol, currently it relies on the XMPP \cite{76} protocol. It is still under active development and in use.

The goals of OurGrid system partly overlap with goals of our system. It also tries to harness idle computing power, and it tries to attract users by allowing them to use cluster power in exchange for their donated power. Similarly as our system, OurGrid system is focusing on making joining of new users as easy as possible. Identities of nodes are also based on self issues certificates.

The main difference between OurGrid and our system is in a layer of implementation. While our system supports all types of applications, OurGrid works only with applications specifically written for that system. OurGrid supports only remote execution, but it does not support preemptive process migration. OurGrid only distinguishes trusted and non-trusted nodes, while our system provides a fine grained security mechanism where user can control access permissions on per node bases.

2.2.10 Virtual organization clusters

Virtual organization clusters (VOC) \cite{5} solution shares some aspects with our work. In VOC, cluster users are dynamically provisioned with resources as they need them. The resources are provided in a form of equally configured virtual machines. The virtual machines based approach has an advantage of easy provisioning of homogeneous resources, that are often required or at least expected by cluster jobs. Another possible advantage is in higher security of host computers, as they are isolated from cluster task by virtual machine.

The main difference between the VOC and our work is the level of abstraction provided. The VOC solution provides transparent addition of machines for a clustering solution, but it is not itself a clustering solution. There has to be some other existing clustering solution running on the environment provided by VOC. Contrary, Clondike is a real clustering solution and it does both dynamic resource allocation and clustering functionality. In theory, Clondike could be used as one of the clustering systems in VOC solution.

If a non secure clustering solution is used with VOC, it does not guarantee much additional security. The virtual machines could help protect host computers from immigrated remote tasks, but they cannot protect the task manipulation by host machine owners. A similar security system as we have proposed for Clondike could be used for VOC to give users an opportunity to protect against malicious job modifications.

2.2.11 BonjourGrid

BonjourGrid \cite{4} is a relatively new project with partially similar goals to ours. The system provides fully decentralized desktop grid, where each user can form his own custom distributed environment. The actual execution environment is not provided by the
BonjourGrid, but rather by other existing system. Currently, the system supports only
XtremWeb [33], but Condor or BOINIC support are planned for future versions.

Unlike our system, where we paid a lot of attention to security, the BonjourGrid does
not consider security as the main topic and it is rather a future work. Also, by using
the other systems to provide the execution environment, BonjourGrid remains limited to
their capabilities, so for example process migration support will be severely limited. The
scheduling algorithm used in BonjourGrid is not clear, but it seems it assumes machines
obey the agreed protocol, which is not a secure assumption in distributed environment.

2.2.12 Plan 9

Plan 9 [72] is a notable example of a distributed system build from scratch. It is based on
an idea of uniform resource sharing, where each computer can expose any of its resource to
other using a common 9P protocol. Each user have its own view of the system, depending
on which resources he maps to his private namespace. While the system is not used in
practice, the 9P protocol itself is still used and for example Linux features a distributed
filesystem based on this protocol.

Plan 9 system can run only programs specifically written for this operating system,
while our system can run unmodified Linux binaries. Plan 9 has a build in security system,
but it relies on a centralized security services. Plan 9 does not support preemptive process
migration, non preemptive migration is supported in form of mapping a remote resource
and executing a process there.

2.2.13 Clondike

Finally, we mention the Clondike system on which is our work based on. The Clondike
system was designed and implemented by students of the Czech Technical University as
an innovative system that should allow secure resource sharing in clusters while preserving
SSI features. An extensive research by Martin Kačer published in [46] compares the system
to all major contemporary clustering systems and argues for a need of a new system.

Clondike is so far the only SSI clustering system that does address security requirements.
It was the nearest system to our envisioned system so it was a natural decision to base our
work on Clondike. The main functional difference between Clondike and our envisioned
peer to peer cluster was in a responsibility of each nodes. In Clondike a single node played
role of a cluster coordinator, while all other nodes could be used only as worker nodes. In
our system, each node can form its own cluster and use any other node as its worker.

The subtle difference in role of nodes have a major impact on the system characteris-
tics, but as well requirements on its components. In contrast to the original system, our
system does not have any single point of failure, neither any single centralized performance
bottleneck. On the other hand, the system becomes technically more complex, as it cannot
rely on any centralized services and algorithms as the original system did. We discuss in
Chapter 3 how to address new challenges of such a decentralized system.
Chapter 3
System Architecture

Computers or workstations participating in clusters are typically called nodes. In dedicated clusters, all nodes belong to the same administrative domain and share the same filesystem, user space, and process space. This is quite limiting for workstation clusters, because all nodes have to give up their autonomy. To overcome this problem, non-dedicated clusters are sometimes used. In a non-dedicated cluster, there is a distinguished node, often called the core or master node, that is responsible for launching cluster jobs and all other nodes offer their spare computing resources to the core node. The other nodes, called detached or worker nodes, remain fully autonomous and they keep the processes from the core node isolated from local processes.

Even though non-dedicated clusters can be securely used in workstation clusters, there are 2 main limitation with non-dedicated clusters. First, the core node is both a single point of failure and a potential performance bottleneck of a cluster. Secondly, the owner of participating workstations cannot directly benefit from being part of the cluster. If they want to use the clustering functionality, they need to have an account on the core node and start their cluster jobs from there.

In our proposed architecture, we address the deficiencies of non-dedicated clusters by a significant modification to the way how the clusters are formed. We allow any participating node to act both as a detached and as a core node. This implies that any node can initiate a cluster of its own and use the other nodes as its detached nodes. A node can serve both as a core node of its cluster and as a detached node of another cluster at the same time. In addition, a node can as well serve as a detached node of multiple clusters and keep those clusters isolated from each other. Figures 3.1, 3.2, and 3.3 illustrate differences among dedicated clusters, non-dedicated clusters, and our proposed architecture. Because of the nature of our architecture, where each node can form its own non-dedicated cluster, we call the system overlapping non-dedicated clusters (ONDC).

We distinguish two main states of detached nodes (per cluster):

**Active** A node is performing some remote computation originating from the core node of its cluster.

**Passive** A node is not performing any remote computation from the core node, but is still
Figure 3.1: A schematic figure of a dedicated cluster. All nodes are equal and form a single cluster.

a part of a cluster and is prepared to accept new processes from the core node.

The state holds per cluster, so if a detached node is participating in multiple clusters, it can be active in some clusters and passive in others. Nodes in the passive state do not consume any significant resources of the cluster, they should just periodically exchange status information and be ready for upcoming processes. A node can in theory by active for multiple clusters at the same time, but this could easily lead to overloading of the node, so a scheduler should try to keep a node active for a very limited number of clusters at a time. Limiting of the number of clusters where a node acts as a passive node is not necessary, since a passive participation is not resource demanding.

If there are $N$ participating nodes, ONDC can in theory have up to $N$ clusters, each of them with up to $N$ nodes. Naturally, this does not increase the computing performance of the system to $N \times N$, most of the detached nodes will either remain in passive mode or will be loaded with remote jobs, and the individual jobs will be processed slower. Nevertheless, the motivation for enabling clusters overlap is not in increasing computing power, but rather it should make the system more tolerant to performance bottlenecks and single-point-of-failures. In an ONDC, there is no single point of failure, a failing node results in its own cluster termination and it can possibly affect another cluster where the node was active. As we mentioned, the node should be active for a very limited number of clusters. In addition, some applications can automatically survive a failure of a detached node via checkpointing, see Subsection 4.4. Also, the core node is no more a performance bottleneck of the system, as any number of nodes can act as core nodes, so there may be many independent cluster computations running at the same time, initiated by different nodes and ideally using different detached nodes.

In addition to eliminating single point of failure and bottlenecks, the ONDC is more attractive for users than other dedicated or non-dedicated clustering solutions. Since each
3.1 Full decentralization

Any centralized component or service in a distributed system is a potential problem for scalability and fault tolerance. The scalability is not necessarily a problem in a scale of a few hundred nodes, but fault tolerance must be addressed even in smaller clusters. A solution to fault tolerance could be replication of components with stand-by nodes ready to act in case of a primary component failure. However, the system should work for any pair of nodes as long as they remain connected, no matter what is the state of other nodes or remaining network. This would mean that every node has to host its own instance of cluster component/service backup.

Another important argument for decentralization is trust and ease of joining of new participating node can act as a core node, it can directly benefit from participation in the system. This could be a strong incentive for users to participate, because they can get something back from being part of the cluster. There are proposals of scheduling systems that reward users for contributing their computing power and in exchange offering them cluster computing power back [54, 19, 64].

Our proposed architecture is similar to some existing clustering solutions aimed at building clusters from workstations [9, 62]. However, unlike those systems, our system is designed with recognition of trust problems and as well its architecture is designed to satisfy requirements of large scale systems described in Section 1.2. In the following sections, we describe how the key cluster features could work in ONDC, we focus especially on security and scheduling.
CHAPTER 3. SYSTEM ARCHITECTURE

Figure 3.3: A schematic figure of an overlapping non-dedicated cluster. Each node forms its own cluster and acts as a core of that cluster. Each node acting as a core node can use other nodes as detached nodes of its cluster. Each of the nodes acting as a detached node hosts an internal nested virtual environment for each other connected core node.

nodes. Any centralization of services requires nodes relying on this service to trust that service. Similarly as nodes currently participating in a system do not trust newly joined unknown nodes, the new nodes should not be required to trust any of the cluster services. In addition to trust problem with service hosting, it may be also problematic to decide which node should host them, as the node hosting a service would likely suffer some performance hit without any benefit. In general, owners would try to avoid hosting shared services unless they have malicious intentions.

For all the reasons above, we decided to have a system fully decentralized with no centralized components. We pay a special attention to designing all system components, to obey the full decentralization principle. Most importantly, all subsystems should avoid the need of having some cluster service hosted by other nodes.

3.2 Process migration

In order to use remote nodes computing power, clusters need a way to execute some processes remotely on those nodes. The mechanism to achieve that is called process migration. There are 2 main types of process migration - preemptive and non-preemptive. We describe these basic types of process migration in Section 2.1.3.7.

Non-preemptive migration can be generally faster, as it does not require a process address space to be copied, but preemptive migration is more flexible, as it gives to a cluster scheduler an opportunity to react on changes in the environment. An ability to dynamically react to environment changes is especially important in clusters build from
workstations. The workstations are typically used by cluster when the workstation user is not actively using his workstations. However, when the user starts interacting with his workstation, remote process running on his workstation can negatively impact performance of his workstation. Therefore, the remote process has to be either suspended for the time user is using the workstation, or migrated to another node.

Both types of migration can leave some residual dependencies on the home node. It means that a migrated process is not completely independent of its home node. For example, it can leave there open sockets or open files. In theory, most of the dependencies can be addressed, but it is a technically challenging task. Dedicated SSI clusters try to avoid residual dependencies, since those dependencies decrease fault tolerance, but for the ONDC, it is not a problem, since a termination of a home node terminates its cluster and so all its processes become obsolete.

Our proposed system supports both non-preemptive and preemptive migration\(^1\). Implementation of our migration mechanisms is based on checkpointing and restarting, so in addition to migration it provides as well support for generic rollback for applications without external dependencies. Details of our migration mechanisms are described in [83] and we also provide basic overview in Subsection 4.4.

### 3.3 Filesystem

Simpler clustering solutions could distribute files required for remote computations to remote nodes by some remote copying mechanism, e.g., by `scp`. However, a more general and transparent solution taken by most SSI clustering solutions relies on a distributed filesystem. It is often a key component of the system, its performance and fault tolerance characteristics have a huge impact on the overall cluster characteristics.

Dedicated clusters typically enforce all nodes to have exactly the same operating system image installed (or they boot identical images from a network) and hence they need to share only data required for distributed calculations. Systems build from independent workstations, like the ONDC, have to share also binaries and all libraries required for execution. This is required, because each of the workstations can have its own system installation and they do not necessarily share the same software or the same version of the software.

Despite some promising results in research of serverless distributed filesystems [6, 10], the prevailing model for current distributed filesystems is based on the client-server model. In dedicated and non-dedicated clusters, the distributed filesystem server is usually served by a machine external to the cluster, or by the core node in case of a non-dedicated cluster. In ONDC, each of the nodes could form its own cluster with its own data and execution environment. Therefore, no external filesystem server can be used and each of the core nodes participating in ONDC has to work as a server for its own filesystem.

\(^1\)It does not support preemptive migration directly between remote nodes yet, but it could be easily implemented.
A variety of distributed filesystems exists, but only a few of them are suitable for the ONDC use case. The main problem with most of the filesystems is that they do not allow concurrent access to files from local and remote systems at the same time. This is not required if the filesystem server is external to the cluster and all performed operations are remote. However, in ONDC, the core node exports its own filesystem and so it must be accessed both locally (by the core node) and remotely (by detached nodes). Two most suitable systems supporting those use cases are Plan 9 filesystem [72] and a well known NFS filesystem [79]. Unfortunately, neither of them does address well security and caching. We discuss the integration details more in Subsections 4.9 and 3.4.

3.3.1 Caching

Remote caching of files is essential for effective use of a distributed filesystem. To preserve expected behavior even in distributed environment, the remote caches must be coherent among themselves as well as with the source file (and local filesystem cache on the core node). Caching is traditionally used for read operations, but can be used as well for write operations in either a write-through or write-back style. At least read caching is essential for ONDC, since all binaries, libraries, and data are served by a single core node to all remote nodes and thus the node network interface could easily become overloaded if the data are requested repeatedly. For distribution of binaries and libraries, cooperative caching techniques [13] would be desirable, as this type of caching decreases the work performed by the core node by offloading read operations to other remote nodes.

Plan 9 filesystem does not have any caching of its own and NFS has only non-coherent caching capabilities based on configurable timeouts. In a long term, we would like to provide our own specialized caching layer with advanced features tailored for the system needs (see Section 6.3.1), but it does not exist yet. In order to get a realistic picture of possible performance characteristics of a ONDC system, we have implemented a simple custom caching layer that could be used on top of existing non-caching distributed filesystems. It does not support any advanced features, like write-back caching or cooperative caching and it is not fully cache coherent. The main use of our caching layer is to mount it only over specific folders that do not change frequently and contain required binaries, like /usr/bin, /usr/lib, etc. This relatively simple solution solves the most problematic performance problem in ONDC, the distribution of binaries and libraries required for execution. The solution is simple, yet it does address the most common opportunities for cached data reuse. In a real world usage, it would suffer problems when the local binaries change, as it does not automatically invalidate the caches.

The main difference between our simple caching solution and NFS caching is that our caching layer is independent of underlying filesystem, it could be used both for the Plan 9 and for the NFS with caches disabled, thus achieving a consistent cache behavior across different filesystems. In addition, our cache could be mounted multiple times on subtrees of a single mounted remote filesystem, and in each tree with a different caching strategy.
3.4 Security

Computer security is a very wide term. For users, the most important thing is that an attacker can never access their private data and cannot impact their work in any negative way. There are many possible attacks that a malicious user can try in a cluster environment in order to compromise users’ security. The following is a list of the most important possible attacks implied by participation in a peer to peer cluster:

**Identity forgery** A node claims to be some other node in order to gain an advantage or to get rid of its old bad reputation history.

**Message forgery** A forged message is sent by some node to another, pretending to be from someone else.

**Information forgery** A message is distributed with correct identity, but it contains forged data.

**Migration-based attacks** Attacks made either by sending malicious processes to remote nodes, or by attacking remote processes running on an attacker’s machine.

**Filesystem attacks** Attacks targeted on the distributed filesystem service.

**Eavesdropping** An attacker tries to read network traffic not intended for him.

**Exploiting bugs** An attacker tries to exploit some implementation bug in service to gain control over attacked computer. We do not discuss this problem further, as there is no consensus how to address it best, except for disciplined secure programming, though admittedly it is not a perfect approach.

**Denial of service** A node tries to overload some resource of another node to disrupt its local functionality and make it inaccessible to others.

**Protocol attacks** A wide range of attacks trying to exploit a known deficiency in a protocol. Similar outcomes as the denial-of-service and bug-exploit attacks.

Most of these attacks are ignored in traditional dedicated clusters, since all nodes belong to a single administrative domain. In this case, it is mostly sufficient to protect the clusters from access from outside networks. In a cluster of workstations, each node can be owned by a different administrative entity and so the risks have to be addressed. Ideally, the risks would be addressed transparently by the cluster infrastructure without involving cluster users. Most of the attacks could be addressed in an automated way. However, as we show in Subsection 3.4.3, some migration-related attacks cannot be prevented in a completely transparent way. Whenever an automated transparent solution is not possible, a trust-based security is used. In this type of security, a user has to explicitly define his trust preferences, similarly as he does for example when setting access permissions to his files.

In the following sections, we describe mechanisms required to address cluster related threats.
CHAPTER 3. SYSTEM ARCHITECTURE

3.4.1 Identity

Since parts of the security system need to rely on users expressing their trust, the system must provide some way how users can identify trusted entities. This implies that each entity in the system must have some unique unforgeable identity. Traditional systems often rely on a centralized identity providers, like Kerberos. However, this solution is not suitable for ONDC where no external service providers should exist. A more suitable approach is a fully decentralized solution using self-signed identity certificates. This solution is often used in trust management systems for P2P networks. Those networks are conceptually very similar to ONDC, so we try to base our solution on this field proven approach.

There is some controversy about using identity-based security in large scale clusters. The main argument against using identity-based security is in its lack of scalability, as involving possibly thousands of nodes in access permissions is unmanageable. However, in an ONDC environment, a typical use case is different. An owner of a node will define shared access privileges for all unknown nodes and in a large scale cluster a majority of nodes will be indeed unknown. Then the owner will specify higher access privileges to nodes of trusted owners, but those would be only a small minority of all nodes, typically at most the order of tens of nodes. Since each core node owner is responsible for his own rules setup, there is not too much configuration work per owner and most importantly, it does not grow linearly with the size of a cluster, but rather with the number of trusted nodes.

The self-signed certificates used for representation of the identity carry no semantic information about the owner. They just represent an unforgeable identity proof. The certificates are represented by a randomly generated sufficiently large asymmetric cryptography key pair, where a public part of the certificate is a public key and its hash is signed with a private key. The ownership of a certificate equals to the ownership of a corresponding private key. The ownership of a private key could be verified by standard cryptographic methods. We use a station-to-station protocol for mutual verification of certificate ownership whenever 2 nodes establish a connection.

An advantage of self-signed certificates compared to traditional certificate hierarchy like x509, where a certificate issuing authority exists, is in scalability and easiness of admission of new members. There is no bottleneck in issuing of certificates and new users can join a cluster immediately. A disadvantage of self-signed certificates is that they do not prevent malicious workstation owners in creating a new identity any time they want. This is the price any system has to pay for not having a certificate issuing authority. The problem is mitigated by treating all unknown nodes as equal. So, creating a new identity does not actually bring to an attacker any benefit in the form of trust, except for sweeping away his negative history records.

A single certificate represents a single node and thus indirectly its owner. From the security point of view, there could be a shared certificate for all nodes owned by a single administrative entity. However, it is beneficial to have separate identity for each node, because the identity can be as well used for resource usage accounting, health monitoring,
and connection establishment. If a malicious owner decides to share his identity on multiple of his nodes, it does not compromise security of the system, since he does not gain any privileges he did not have before. It can just confuse other nodes relying on one identity per node, but the confusion would generally lead only to exclusion of machines with conflicting certificates from a cluster, so this is not any significant negative impact.

It would be difficult for cluster users to work with certificates directly, so the system allows users to create meaningful name aliases for certificates and express trust by means of those aliases. So, for example, if a user knows that a certificate belongs to Bob, he can alias the certificate as a \texttt{bob-machine} and then give privileges to this alias instead of a semantic-free certificate.

### 3.4.2 Access control and certificates

The access control mechanism gives users an ability to express who can access their resources. The mechanism should satisfy the following requirements:

- Ease of use.
- Scalability to larger clusters.
- Low impact on performance.
- Flexibility in expressing granted permissions.

To address an ease of use requirement, we try to expose intuitive and simple user interface for the users requiring minimal additional knowledge. Most of the users are familiar with controlling access privileges to their files by setting appropriate permissions. The interface exposed by our access control system is somewhat similar to a standard ACL Linux concept: owners of nodes simply grant permissions to owners of other nodes.\footnote{Typically identified by a locally created alias of their certificate.} The permissions being granted are a bit more complicated than usual ACL permissions as they are not limited only to file access. For the sake of flexibility, the permissions are expressed by 3 generic string values - \texttt{permission type}, \texttt{operation type}, and \texttt{operand}. All those are free form strings and they are interpreted specifically based on the permission type. An example of a permission granting read access to a \texttt{/tmp} folder would be:

\texttt{fs read /tmp}

The permissions are granted by an owner of a core node to other node owners. Evaluation of those permissions is done on the core node.\footnote{Except for special cases when caching is involved, we discuss those in further sections.}

Since each core node owner is responsible only for setting up the access permissions to his node, the complexity of configuration grows at most linearly with the size of a cluster. However, as we mentioned earlier, in a large scale clusters spanning over the Internet, an
owner typically knows only a limited subset of participating node owners and this number
does not necessary depend on the size of the cluster. Since the system allows owners to
express what permissions it grants to unknown nodes, they can easily configure accesses
for all nodes, no matter what is the size of a cluster. Hence, we can conclude that our
system is scalable even to large cluster sizes.

From the performance point of view, the key is that all security decisions can be done
locally without any remote interaction. Remote interaction would add a significant latency
to any restricted operation, so we have to avoid a need of it. In order to do all decisions
locally, a core node needs to have all security related information available, but this is
normally the case, since the core node is gathering all data about processes and about
issued permissions.

In addition to generic format of permissions, we also introduce two additional concepts
to make the system flexible - delegation of permission granting and grouping of nodes.

Delegation  Granting of permissions is normally performed by an owner of a core node.
This works well in typical cases where a node owner administers just a single node. In
case an owner has multiple nodes, he would be required to configure each of these nodes
separately. With the support of permission granting delegation, an owner can configure all
his nodes to delegate all privileges to a single node and from this node he can configure
all his other nodes. This way, the configuration is centralized from the point of a single
user. It does not limit scalability of a cluster itself, because the size of this centralization
depends only on a number of machines owned by a single user, not on the cluster size. In
addition, it is only an optional feature and node owners are not required to use it.

In order to represent delegations and delegated permissions, a secure unforgeable rep-
resentation of rules is required. Granting of remote permissions is done using certificates,
similar to user identity certificates. The following is an example how a permission granting
certificate looks like:

Id: 14
Issuer: <public-key-of-issuer>
Authorizee: <hash-of-authorizee-public-key>
Target: <hash-of-target-public-key>
Can-delegate: true
Permission: fs read /tmp
Signature: <message-signature>

The *issuer* is a node (owner) that issues the certificate and this is also the node that
has to sign the certificate. The *authorizee* is a node being granted the permission and
the *target* is a node where the permission can be executed. The *can-delegate* flag specifies
whether a certificate allows its target to delegate contained permission further. A typical
delegation chain can be that node A issues a delegating certificate with certain permissions
to a node B and sets the *can-delegate* flag to true. The node B can then issue another
permission certificate to a node C, with a subset of permissions granted from A and target
set to the node A. The node B can as well grant delegation privileges to node C, thus further extending a delegation chain length, see Figure 3.4 for a simplest delegation chain example.

Figure 3.4: First, node A grants some privileges to node B, including a delegation rights. Then node B grants a subset of those rules to node C. It provides both the certificate with granting of the privileges and a certificate proving that it is allowed to grant such rules.

No matter if the permission was issued directly or using delegation, they are always evaluated directly on the core node to which they are granted (specified by the Target field). For directly issued permissions, all the information is in place. However, for delegated permissions, it may not be the case. In case of the delegated permission, a validity of the permission has to be proven by providing a full delegation certificate chain. As already mentioned, it would be too slow to check permissions remotely, so the core node needs to have sufficient information for evaluation locally. We discuss how to achieve this in Subsection 3.4.2.

Since all remote permission grants are expressed as certificates, we do express also locally issued permissions with no delegation as certificates. This does not have any major advantage except for consistency. The locally issued certificates simply have the can-delegate flag set to False and the Target field is the same as the Issuer field.
**Grouping**  The second feature making the administration easier is the support for node grouping. Any owner of a node can create a logical group of other nodes. This group is local to the creator of that group and it is remote to all other nodes. Local groups allow convenient configuration of non delegated rules, where a core node owner can simply assign a same set of rules to multiple nodes. There is always an implicit **anonymous** group to which all nodes belong. This is the way how a core node owner can grant a common minimum privileges to all nodes, including unknown nodes.

The remote groups are important for delegation. Going back to the case of an administrator owning multiple machines, he can delegate all permissions to a single machine. However, on this machine, he would still have to issue permissions to all nodes one by one. With grouping, he can create a local group of all his nodes on a single node that should serve as an administrative node and issue delegated permissions from all his nodes to that administrative node. Figure 3.5 illustrates this use case. For the nodes, the group defined on the administrative node would be a remote group, since it was not defined by them. In order to evaluate the delegated rule access, the nodes need to know both full delegated chain of granted permissions, but also a proof that they belong to a remote group that is a subject of the permission. This proof is again issued in the form of certificate, this time a membership certificate.

Figure 3.5: First, owner of nodes A, B, and C delegates all permissions from nodes A and B to node C. Then, from node C he makes a group G1 containing both A and B. From then on, he can grant permissions to the whole group at once.
3.4. SECURITY

Revocation Sometimes, it is required to revoke granted authorizations, e.g., when it is known that a trusted remote node certificate was compromised. It is easy for permissions granted directly from the core node; because each permission is represented by a certificate, the core node can just delete the locally held certificate representing the permission. In case a delegation was used, the revocation has to be communicated remotely. When an issuer of delegated permissions wants to revoke it, he issues a special revocation certificate referring to original certificate granting a permission. This certificate needs to be communicated to the core node as soon as possible, so that it knows about revocation. Group membership revocation works in a similar way.

Storage and distribution of certificates Any distributed system using decentralized certificate-based security mechanisms has to solve 2 problems related to certificates: where are they stored and how do the nodes find right certificates when they need them. Often a centralized certificate store is used, but this is not a good choice for the ONDC, since it would undermine all advantages of having fully distributed security system. P2P systems sometimes use distributed hash tables as certificates storage \[25\] with distributed algorithms for location of certificates based on search criteria. This is not a good solution for ONDC, because there are different assumptions and requirements on availability. In P2P networks, if a node gets disconnected from the main network, it will simply stop its participation in the network. However, in ONDC it is not required that all nodes are connected. In fact, if there are just 2 nodes connected to each other, they should be still able to interact no matter if they get disconnected from all others. If they rely on certificates being in global distributed hash table, they may be in a situation they cannot evaluate each other’s permissions.

For ONDC, we propose a special purpose storage and distribution mechanism matching well the environment characteristics. To address the storage problem, we need to focus on the fact that any pair of nodes should be able to obtain all the required certificates to evaluate its peers access privileges. This must hold even if the pair of nodes is disconnected from all other nodes. This requirement could be satisfied by the following rules:

1. All locally issued permissions and group membership information are held on the node that issued them.

2. When a node is granted a delegated permission, it stores the certificate with this grant plus all other certificates up in the chain to the core node.

3. When a node is assigned to a group, it stores the certificate about its membership.

4. If a node A has some delegated privileges and if any authorizee or issuer of any of the certificates in the chain is a group, then a group membership of the issuer must be stored on the node A.

5. Revocation certificates are kept on the node that issued them and on the core node. After the core node confirms that it got a revocation certificate, the issuer can delete it and it can also delete the certificate being revoked.
The first rule is natural, since core nodes need to evaluate all directly issued rules themselves. The second rule ensures that a simple delegation without grouping always works. The holder of the delegated rule keeps a whole delegation chain locally, so he can prove his permissions by providing this chain to the core node when required. The 3rd and 4th rule is required in case that a group membership is used in combination with delegations. The authorizee of the delegation could be a group defined by the delegating node. In this case, the 3rd rule is required to prove that the authorizee is indeed a member of that group. The 4th rule is required to verify a delegation chain that has some of its authorizee or targeted specified as a group. Such a chain could be verified only when all membership certificates are known. Figure 3.6 captures the following example in which the 4th rule is required:

1. A node A issues a delegating certificate to a node B.
2. The node B creates a local group G that among others contains a node C.
3. The node B grants delegating permissions to members of the group G.
4. The node C, as a member of the group G, grants a permission to node D.

Figure 3.6: When the node C grants some privileges to operate on A to the node D, the node A does not directly know the certificate issued by C is valid. It can know this only if it sees a delegation certificate from B granting permission to the group G and also a membership certificate issued by B assigning the node C to the group G.
In order to validate the chain above, the node A needs to see all delegating certificates, but also the group membership certificate issued by the node B about group membership of the node C. Without that certificate, it cannot know that the certificate granted by the node C in step 4 is valid.

In case a permission is not revoked directly by a core node, but rather by some node in the middle of delegation chain, the revocation has to be communicated back to the core node. The fifth rule ensures the core node is eventually informed about the revocation. Until the core node gets the revocation certificate, the certificate must be kept somewhere, so up to that time it is a responsibility of the issuer of that revocation certificate to store it.

In addition to the storage of full certificate chains on their authorizee nodes, we also allow an optional back propagation of certificates backwards in a chain, towards the core node. This means that a delegating node will also store all certificates that were issued by nodes relying on authorization granted by that delegating node. The mechanism is recursive, so, for example in the situation captured on Figure 3.6, the node A would store locally a permission granted by B to the node C and also a permission granted by the node C to the node D. The node B would store locally the permission granted by the node C to the node D. Image 3.7 illustrates the propagation of certificates along with the back propagation.

It is not a necessary feature. By propagating the certificates back towards the core node, the system allows node owners to see a status of delegated permissions at the time. Without that, the node owners have no way to know what permissions were actually issued, unless they probe remote nodes for data. In case of large delegating trees the delegating node and especially the core node could become overloaded by accumulated rules so the
back propagation is only optional.

From the rules above, it is clear that every certificate is stored on at least one node, i.e., its issuing node. Most of the certificates are also stored on nodes being authorized, which at most doubles the storage requirements. Delegation storage requirements are not bounded though. In theory, a delegation chain can be as long as is the size of cluster, but this is not a realistic case. Generally, we can expect very limited depths of delegation, the maximum delegation depth could be even enforced by the system if required. Let $K$ denote the maximum depth of delegation chains. Then the storage requirements would be at most $K \times K$ (if the back propagation is used). Since $K$ does not grow with the cluster size, the storage overhead is still constant. For example, if we strictly limit the depth of certification to 4, the overhead for delegation, defined as number of times a single certificate is stored, would be 16. Without back propagation the overhead is reduced to half. The total overhead is in either case under 20.

All certificates that have to be stored on multiple nodes are exchanged immediately after being issued. It is possible that some of the nodes that have to store a certificate are not connected to the cluster at the time of certificate creation. It is a responsibility of the certificate issuer to retry later and ensure that eventually all required owners get the certificates.

If the delegation is used and no back propagation is enabled, a core node does not have all certificates it needs to evaluate permissions. In addition, if a grouping is used, it may miss some group certificates required to verify delegation chain. The reason for not storing all certificates on the core node is just in limiting storage requirements on core nodes as large delegation chains could possibly overload core nodes. In order to be able to evaluate access requests depending on missing certificates, the core node needs to get those certificates at least temporarily. Providing them exactly at the time when they are needed could have severe negative performance implications, as it would add network latency to evaluation of access permissions. Instead, it is a responsibility of the node that wants to execute the permission to proactively push all required certificates to the core node. If the remote node obeys the rules for certificates storage, it should own all required certificates. The certificates are pushed to a core node when a secure connection between the core node and a peer detached node is established. The core node keeps those certificates temporarily only for time when the connection with peer lasts, but it throws away those certificates after the connection is terminated.

The described delegation and back propagation mechanism gives node owners unique control over the system. They could choose a local access control system, fully decentralized access control solution, or a mix of those two. By back propagation flag, they can control certificates storage requirements on their nodes, without affecting functionality of the system.

---

4Provided they will ever again connect to the cluster.
3.4. SECURITY

3.4.3 Migration security and stigmata

The ability to migrate processes introduces a unique security threats into the system. Two classes of attacks can be performed. The first type of attack can be performed by an attacker from a core node, migrating a malicious program to a remote node in an attempt to get control over that node. This attack can be prevented by running all processes on remote nodes with very limited privileges. Due to the ONDC nature, processes from different core nodes running on the same node should run with different low privilege accounts, otherwise they could interact with each other\footnote{This is currently not enforced by current implementation, but it is just a small technical change.}.

A more severe type of attack can be made by an owner of a detached node. Since an owner of a node has superuser privileges to his machine, he has a full control over processes running on his node. He can read the content of memory of remote processes immigrated to his machine, but he can also alter their memory content, even the programs themselves. Detecting such alternation of programs or of data is an extremely difficult problem in general case and we do not try to address it. Instead, we delegate this decision to cluster users to express their trust in other users by means of the access control mechanisms. All sensitive operations, such as file access, sending signals, or opening sockets, should be guarded by access control mechanisms. Whenever a sensitive operation is attempted, an access control mechanism is consulted and operation is either permitted or rejected, based on user settings and on the program being executed.

It is not sufficient to enforce per node access restrictions only when a process is running on that node. Since an owner of a malicious node could modify the running program, it can try to perform a malicious operation later, when the program is running on a different node, or it can make use of some sensitive data read earlier. There are 3 types of threats related to possibility of modifying the program:

Current node threat This is a risk that the owner of a node modifies a program and tries to execute this modified code on his node.

Past node threat A risk that a node owner modifies a program and then the program is migrated away. The malicious owner’s node becomes a past node and a new execution node is executing possibly maliciously modified program.

Future node threat A program can run on some trusted node and read sensitive data. Then it may be migrated away. If its future node is untrusted, it may get access to sensitive data read into memory of the process.

A stigmata \footnote{This is currently not enforced by current implementation, but it is just a small technical change.} mechanism was proposed to address the security risks associated with migration. To address the current node threat, it is sufficient to give users an option to restrict privileges of remote nodes. This way, the malicious owner of remote node becomes restricted of what he can achieve by modifying a process. To address the past node threat, the system must keep track of all visited nodes and whenever a node requests some access privilege, it must be satisfied for all nodes visited during the process lifetime.
When enforcing this rule, a malicious user cannot achieve any more advantage than in current node threat, because migrating process away cannot ever increase access privileges of the process. To prevent a future node threat, the system must keep track of restricted operations performed by a process that may give future nodes access to some sensitive data. Examples of such privileges are filesystem operations or working with sockets. On the other hand, operations like killing of another process is not relevant to the future node threat, as it is executed immediately and does not have any consequences on a future security. The list of performed secure operations has to be consulted whenever a process is being migrated to a next node and that node has to have permission to execute all of process past operations. A more detailed explanation of idea behind stigmata could be found in [43].

Since the future node threat requires keeping list of all performed operations for running processes, it could become too expensive in terms of performance. It is not strictly required to keep track of those data for all processes, it has to be done only for those processes that can eventually be later migrated. For some processes, it is clear they are not going to be ever migrated, while for others it may be unclear. A scheduler can make speculative decisions for unknown processes and track for some and ignore tracking for others. Those without tracking will be then excluded from migration and will stay all time on their home node.

3.4.4 Stigmata and caching

There is a complication associated with possible performance optimizations through remote caching, most importantly remote filesystem caches. If the security check results are cached on remote node, the decision mechanisms will be unaware of process stigmata, the cached result will just reflect access permissions specific to that node. This is the case of filesystem caches, where operation results are cached and this could violate stigmata rules. To illustrate the problem, we use the following example:

1. A process is migrated to a node A. The node A does not have access to a sensitive-dir. If the node A tries to access this directory, the access would be rejected. However, the node A can modify process to perform some malicious actions in the sensitive-dir and then migrate it home.

2. The core node decides where to migrate the process after it is migrated home. If no rule prevents it, it can be migrated to a node B, that has access to the sensitive-dir.

3. The process is migrated to the node B, tries to perform a malicious operation in sensitive-dir. If the request is performed in a standard way, i.e. it is delegated to core node, the security mechanisms would detect stigmata rules violation and would reject the operation. However, it is possible there was some other process migrated to B before, and performed a legal action in sensitive-dir and a portion of filesystem remained cached. If the operation performed by malicious process migrated from the node A performs its operation only on locally cached files, it would be a violation
of stigmata rules. Write operations can be later negated when a modification is eventually propagated to the filesystem server, but reads violations may never be detected.

There are several possible ways how to prevent this security problem:

1. Stay away from caching and propagate all operations to the core node.
2. Use caching for data, but still propagate all security checks to the core node.
3. Use caching, but perform stigmata checks locally on detached node. This would require propagating all stigmata information to the execution node of a process.
4. Restrict eligible targets of migration to nodes, that do not have wider privileges than any of the (remote) nodes visited during process lifetime. This way, it is guaranteed a process cannot get to a node that has cached some data that a process is not supposed to access.

Due to the importance of caches on performance in ONDC, the option 1 is clearly unfavorable. The option 2 has a lower performance penalty, as it would be sufficient to perform checks only on certain operations. For example, for files it would be sufficient to perform checks on file opening and later reading/writing could be performed in a local cache. The option 3 avoids a penalty for remote checks altogether by delegating those checks to a detached node. To see why this does not undermine security of the system, imagine we have a node owner B, who does alter cluster code on his machine to ignore the delegated checks. This way, a process may violate stigmata mechanism, but the violation can affect only files to which an owner of B got access from a core node owner. Even if the owner of the node B does not alter security mechanisms, he still has access to those files, so he can simply distribute those files further to other users. This implies, the overall file security is still same even if we delegate cached decisions to remote node owners.

The option 4 has a huge advantage in technical realization. The only thing required to implement this rule is a proper restriction in scheduler, which is much simpler compared to changes in filesystem code required by the options 2 and 3. The option 4 may seem too restrictive, but in a real world case it is quite common to have a large set of nodes with equal privileges. In such a scenario, it is typically easy to find other node with equal set privileges and migrate a process to it. Currently we are using the option 4, but for performance reasons the option 3 may be advantageous in a future.

3.4.5 Scheduling with stigmata

Stigmata mechanisms introduce a special requirement on scheduler. Ideally, the scheduler would know what types of access permission will process need during its lifetime and

---

6It would be good to have the option 3 as an optional choice, as it naturally brings a small risk of temporarily staleness of security data on remote nodes.
schedule it to run only on nodes with sufficient access privileges. The required access permissions are, however, often not known upfront and so the scheduler can sometimes only guess. If a violation of access occurs, the process can be either terminated or rolled back, depending on type of its external dependencies. When a rollback is performed, the scheduler should never attempt to migrate the process again to a node without permission to perform violating operation. For this, a forward stigmata mechanism has been proposed. The forward stigmata contain a set of restricted operations that violated some access permission during a process running time and caused rollback. The scheduler should consult forward stigmata when making decision where is the process going to be migrated, so that no predictable conflicts occur.

### 3.4.6 Filesystem security

Besides standard security problems addressed by most of distributed filesystems, ONDC architecture has additional security requirements. The core node should provide its files to detached nodes, but it does not know up front which file to what nodes it will provide. If there is no integration between filesystem and ONDC, it would require the filesystem server to open access to all nodes to all possibly required files. In order to reduce visibility of files, integration with cluster security mechanism is required.

The security system has to be consulted on every file access, whether a requesting node has access permission to a file or not. To be able to make this decision, it needs to know an identity of requesting node, so when a node connects to the filesystem, it has to present its certificate and proof its identity. The certificate is exchanged and verified only at connection establishment time and later the connection is trusted to belong to a verified peer node. It is generally not easy to break undetected into an existing TCP/IP session and if that should be a problem, a IPSec channel between nodes code be established. On each file open operation, the detached node provides process identifier on behalf of which the operation is being performed. This identifier combined with verified node identity is checked by the system. First, the process must reside on the requesting node. Second, the access request is checked against the process stigmata and it is either accepted or rejected.

### 3.4.7 Data transfer security

There are common techniques how to protect network channels from possible attacks and in theory any of these could be used for security channels used by an ONDC cluster. We discuss some options for local and remote networks in Section 3.6.

Securing of channels should be optional as it carries some performance penalty. In many environments it is not necessary to use security on channels as the network can be considered safe. This could be for example a computer laboratory in school, where users could trust administrator of network (i.e. school), but not to owner of other computers connected to the network.

A decision whether to secure channel or not could be left to user, but since we do not want to place too much technical requirements on users an implicit defaults should be
set as carefully as possible. The best compromise between performance and security for defaults is making no security on local network Ethernet segments and secure in all other cases. Marking implicitly local network as trusted opens a possible security hole in case an attacker has direct control over the network switch used by the network segment of the node, but in typical environments this is not the case. In all other cases there may be multiple network components involved and the system should conservatively assume no trust.

3.4.8 Denial of service

Our system in its current version does not provide any protection against denial of service attacks. However, the present infrastructure with unforgeable identities gives an opportunity to prevent some denial of service attacks. First, a node could detect excessive amount of requests originating from a remote node with the same identifier. A more complicated case is when a remote attacker uses a weakness of self signed certificates and generates a new identity for each request. In this case, the detection is more difficult, but still possible. The system could in addition to identity track as well a network characteristics of the requests, and if there are for example too many requests coming from a single IP address, than this IP address could be blocked. Blocking of IP address may result in denial of service for some legitimate users on this IP address who did not participate in the attack. To lessen the impact of this complication, the node under the attack could distinguish requests coming from the suspicious IP and allow selectively requests with trusted identity certificates and block all others.

3.4.9 Handling unknown nodes

An important decision is how to handle unknown nodes, i.e. nodes whose certificate is not known to the node (owner) interacting with it. The safest option would be to completely ignore/forbid any requests from unknown nodes, but that would severely limit growth potential of cluster. No new user would be able to join without knowing some existing user of a cluster. The other option is to allow any access to all nodes and restrict access to known nodes only when using some sensitive private data. This is certainly more scalable option, but it posses a risk that less experience users would ignore this step and may get their data compromised.

In our solution we choose a second option despite its risks. The system should provide sensible default configuration, restricting access to files that should always be restricted (like passwords file) and possibly offer users on installation time an option to choose a folder where globally accessible files used by cluster jobs should reside.
3.5 Scheduling

Schedulers in traditional dedicated clusters can make sophisticated planning decisions in order to achieve best utilization of the cluster. When deciding placement of new remote processes, they can usually consult a global trusted information about cluster state, they precisely know performance characteristics of each participating node and there is usually just a single (possibly distributed) scheduler making the decisions so no unexpected processes appear in the cluster.

The situation is very different in ONDC environments. ONDC is meant to scale to large number of nodes and so a global state information would be a scalability limiting factor. Moreover, the nodes do not have any implicit trust among each other and so the information about state of remote nodes is not trusted. Similarly, performance characteristics of nodes can vary and the claims about those characteristics are possibly false. Each node can serve as a workstation for its owner and so a new high priority processes can be start unpredictably on those node. Finally, similarly as with the security ONDC should not rely on any external scheduling service and each cluster should be scheduled by its own core node. So each of the core nodes have its own scheduler and those schedulers do not implicitly trust each other neither have information about scheduling decisions of others.

In the environment where most of the information about system state is either missing or untrusted and many unpredictable events can happen at hard to predict times, it is difficult to design a scheduler that would optimize resource usage for maximum efficiency. Indeed, the efficiency itself is different than in dedicated clusters. In dedicated cluster, the schedulers typically try to schedule so that a computation is finished in a shortest possible time. However, in ONDC each of the core nodes has its own selfish needs and can try to offload as much local computation as possible without offering anything to the cluster, a problem commonly known from P2P networks as a free rider problem [51].

Not only the environment makes it very difficult to design efficient scheduler, but each type of application may need a different approach. For example the easiest type of application, a bag of independent processes, can be scheduled relatively easy as all processes are independent, a more tightly coupled applications may benefit from coscheduling [36] techniques. Due to the problems with designing a good scheduler that would satisfy all types of applications in ONDC environment, we paid a special attention to make introduction of new schedulers and experimenting with them as easy as possible. It is also possible to use different schedulers for different types of calculations, see details in Subsection 4.7.

One promising option for scheduling strategies addressing a lot of complications of the environment are market inspired schedulers [54] [19]. This type of schedulers was studied in many environments where a lot of untrusted parties compete for computational resources. However, existing solutions rely on an external bid matching components used by individual participants and since in ONDC no external service should exists, the solutions are not directly applicable and a further research on suitability is required.
3.5. **SCHEDULING**

3.5.1 **Used algorithm**

We define a term **cluster task** as a task, that could be potentially migrated by the cluster. Those are generally tasks, for which migration would be efficient, like any long time running CPU intensive computation, or even short duration but CPU intensive tasks, like **gcc** compilation. Such tasks could be detected either by a static predefined regular expression-based rules about their names, or by a more sophisticated prediction-based technique, see Subsection 3.7. In either case, they can be marked by a special classification to make a scheduler aware of them.

In clusters with support for preemptive scheduling, even a bad scheduling decisions for long running tasks are not a major problem, because the system has always an option to adjust incorrect decision based on observation of a calculation progress. Therefore, in algorithms used in our ONDC system we have paid most attention to a well balanced placement of tasks during a non-preemptive migration, combined with corrective actions performed when required. An important objective of the algorithm is to limit the number of locally running cluster tasks on a core node, since the core node has to handle all support functionality like placement of new tasks or serving of files, and so it can easily become bottleneck of a calculation. On the other hand, not all tasks should be migrated away, because often a task can be executed faster locally and if a core node is underutilized, the overall execution time may be worse.

Finding the best scheduling algorithm for the ONDC environment was not a focus of our research and it is rather a follow up research. Nevertheless, to verify usability of ONDC platform some algorithms have to be proposed and used for testing. Experimenting with algorithms in our system is simple thanks to its design (see Section 4.7) and so we have evaluated a couple of existing algorithms and tuned them for our system.

A scheduler used in our solution is simple, yet does deal well with environment for some types of calculations as demonstrated by experiments, see Chapter 5. It is remotely based on the scheduler used in the P2P grid OurGrid [64], but modified for our system. The algorithm consists of two separate parts - non-preemptive decisions and preemptive decisions.

The non-preemptive decisions algorithm is run always when a new process is started. The decision process is as follows:

1. If the number of locally running cluster tasks is lower than a threshold, keep the new task on the local node.

2. Otherwise find the least loaded remote node, that is willing to accept a new task and try to emigrate the task there.

3. If no remote node is found, the task is kept on the local node.

---

7OurGrid does not support preemptive process migration like our system, so its scheduler cannot easily intervene when load imbalance is detected.

8More precisely, when an **exec** command is executed.
A threshold used in point 1) was experimentally determined to be the number of CPU cores + 1, though the number may differ depending on a type of calculation. In order to determine a load of a remote node, a simple task execution node tracking mechanism is used. The tracking is only local, so it tracks only tasks initiated from a node where the scheduler is running. This is sub optimal, as there could be other tasks running on remote node, but an advantage of this approach is it does not require any untrusted information from remote node. As our measurements (see Section 5.2) show, it is not a limiting factor for bag of tasks type of calculations as the algorithm is self regulatory - if some node is overloaded it is finishing its tasks in a lower rate and hence it gets less tasks to calculate in total. The self regulatory aspect of the algorithm makes the algorithm oblivious to durations of individual tasks emigrated to remote nodes.

As mentioned in point 2), the scheduler considers willingness of remote node to accept a task. This is in fact only a hint as this information is remotely propagated from remote nodes as a part of regular status broadcast. The information may be stale and an actual decision whether to accept a task or not is made only at the time task migration is initiated. At this time, a remote node decides if it accepts the task or not. The decision based on a local node load is made and it does as well consider security restrictions placed by user, i.e. for example user can ban certain cluster users from using his workstation.

The preemptive migration algorithm is executed at 2 different times. A simple algorithm is executed when some local interactive user initiated tasks\(^9\) are started on a detached node and it is at the same time running some remote tasks. In order to prevent interference of remote tasks with local user work, the remote tasks have to be suspended and migrated away. Since our system does not support direct migration between remote nodes yet, the tasks have to be send home to their own core nodes. The migration home is not performed immediately, but rather with some delay (10 seconds) if local activity is continued. This prevents a short temporary local activity to emigrate all remote tasks away and it also gives the short duration tasks, like compilation, some time to finish, instead of being expensively migrated.

The second part of preemptive scheduling is checking for too many cluster tasks running on a core node and it tries to emigrate some tasks away if it detects overload. The algorithm is run periodically with predefined interval (1 second in our tests):

1. Check if the number of locally running cluster tasks is higher than a threshold.
2. If so, try to find remote nodes for exceeding number of tasks.
3. Preemptively emigrate all tasks above the threshold for which a remote node was found.

The threshold of local cluster tasks and the algorithm to find a best remote node are same as in the non-preemptive algorithm. The main purpose of this algorithm is to prevent overloading of core node, if possible. The situation when the core node contains more then

\(^9\)Tasks like a browser, development IDE, or rendering program.
required number of tasks can happened in 2 possible ways. First, when a new task was executed, there were no (free) remote nodes available and so it had to be executed locally even though migration would have been preferred. A second case is when some tasks were migrated back from remote node as described above.

Because of its self regulatory aspects, techniques like the one we used are sometimes referred as self scheduling. The scheduling strategy itself does not address the free rider problem, but it could be easily extended by accounting of donated usage to node identifiers and when deciding between multiple nodes the nodes with better donation history could be preferred.

3.6 Network interconnection

There is a huge amount of protocol and topology options for building computer networks. In our system we currently use a combination of UDP and TCP protocols as described in more detailed in Section 4.3. UDP packets are used for broadcasting of node presence and information. As long as the nodes are members of a single network segment, all nodes could find each other by this way. Nodes keep track of a limited number of other nodes and keep an active connection to even a smaller number of remote nodes. There is a predefined minimum of nodes that should be connected at the time and more nodes are connected only if required. Nevertheless, having a connection with other node is not resource expensive, so in clusters with less than 100 nodes, each of the nodes can easily connect with all others.

Our mechanism works well in a single network segment clusters, but it has some limitations in a larger scale:

1. At some point there could be too many nodes to connect with each other and some smarter selection criteria may be required to choose a subset of nodes to connect.

2. A mechanism to find nodes outside of a local network segment will be required for larger scale grid-like deployments.

3. If a local network is untrusted, a secure channel has to be established.

4. If a WAN is untrusted, as it generally should be, a secure channel has to be established.

The problem with limited scalability is not related to single segment clusters with uniform topology; it becomes a problem only when large scale systems are build and the system should try to prefer local nodes over remote nodes. We do not address this problem in our current work. The mechanisms for finding of nodes outside of local segment are practically solved in many peer to peer systems [82, 22, 12], typically they are based partly on static configuration and on some bootstrap nodes. We also do not address these in this work, as it is rather a future step in building a full featured grid system. In the following 2 subsections we discuss options of securing channels both in local networks and in WAN networks.
3.6.1 LAN channel security

Security of data being transferred is a common problem of any distributed system using potentially untrusted network. There is a lot of standard techniques how to address network security and we do not try to invent our own. Instead, we propose a way how to integrate existing systems into our architecture.

An important advantage of our system is that it can exchange short encrypted messages event without any preexisting secure channel. This can be done thanks to existence of identity certificates. A node can encrypt data using public key of a receiver of the message and only the receiver is then be able to read the message. This is not a well performing solution as asymmetric cryptography is computationally expensive, but for infrequent exchanges of short messages it is not a problem. We can take an advantage of this secure message channel to build a more sophisticated better performing channels.

As we show in the implementation Chapter 4 there are several layers of communication in our system and in addition distributed filesystems can use their own communication channels. It would be tedious to secure each of those channels separately, so a common shared solution is preferred. For local network we propose to use standard IPSec [47] security. Since IPSec is running on a low networking layer (Layer 3 of OSI) it can secure all types of communication between cluster nodes.

The basic steps to establish an IPSec channel between 2 nodes are:

1. A configuration on at least one node of a connected node pair indicates a channel should be secured. Since nodes are on a same network segment, the node initiates an IPSec establishment process.

2. The initiating node generates a (secure) random key to be used for the channel. This key is encrypted using a public key of peer node and sent to that node.

3. The sending node waits for confirmation, if it does not receive a confirmation until some timestamp, it does try to resend the key. The receiving node accepts the key and sends a confirmation message.

4. After receiving a confirmation, the sending node configures locally the traffic going to the peer note to use an IPSec encryption using the generated key.

5. After sending a confirmation, the receiving node configures locally the traffic going to the peer note to use an IPSec encryption using the decoded encryption key.

The confirmation messages are required to increase likelihood of successful establishment, as the encrypted messages are exchanged using a UDP channel. Due to a well known limitations of such a confirmation problems [59], it cannot be ever guaranteed that both parties agree and so even with this confirmation step we may get to a situation when one side enables IPSec channel while the second does not. In this case, they cannot effectively communicate and no messages would pass between them. Thanks to the peer dead detection mechanisms (see Section 4.5), both nodes will eventually mark their peer dead. At
this point they revert their IPSec configuration and they should be able to communicate again and they can attempt to establish the channel again.

For dynamic configuration of security channels a Linux interactive command `setkey` could be used, assuming the distribution is using standard `ipsec-tools` library.

### 3.6.2 WAN channel security

IPSec could be used only for computers sharing a network segment. However, an environment more exposed to security threats is represented by WAN networks, where many distinct and untrusted owner of portions of network could gain access to traffic. The argument with single security mechanism for all channel types still holds also in a WAN environment. A suitable solution for this environment is OpenVPN project, that allows securing channels spanning multiple network segments.

OpenVPN allows securing channel either through a certificates based solution or through a pre-shared secret key. The certificates based solution is generally preferred, however our system already has its own certificates and they are not compatible with the certificates used by OpenVPN. Our certificates could be used to securely establish pre-shared secret between 2 nodes similarly as in IPSec case. OpenVPN also supports interactive command line tool that could be used to automatically configure OpenVPN secured channels. There are still several differences from the IPSec case:

1. In WAN network, not all nodes see each other directly, especially due to firewall and network address translation rules. There are techniques how 2 computers could communicate with each other even if they are both hidden behind NAT (see STUN protocol), however OpenVPN does not work with these. We have to assume, that at least one machine can connect to other directly. In addition, we have to assume required firewall ports are open from inside the network (which is a reasonable assumption).

2. Due to a possible asymmetry of connection, only one of the peers could initiate the connection. This problem lies, however, outside of the security discussion as it holds also for non-secured connections. An important technical detail is, that commonly peers could establish a TCP connection over WAN, but bidirection UDP connection may be more problematic. For this reason. Peers connection through WAN network should open an additional direct TCP channel between each other, simulating a UDP channel used in local networks.

3. Unlike IPSec, that does enable encryption on current connection between 2 peers, OpenVPN creates a new secured channel. Therefore, only the TCP channel to transfer encrypted shared secret should be created using the original peer address. The rest of the channels should be created after the secured channel is established and it should use virtual IP addresses assigned by OpenVPN to peers forming this channel.
3.7 Classifications and predictions

In order to provide scheduler with more information about tasks, we propose to use a classification and prediction system. A general idea is, that system will listen on all new process creation or binary execution events. It can extract certain characteristics like a binary being executed, parent child relation, time of execution, or program arguments and environmental variables. Out of these, the system can try to infer some common patterns and mark processes with certain classifications that could be used for scheduling.

An example use of such classification could be predicted execution time based on arguments, and as we show in Subsection 5.5 this could be successful for some types of programs. Other type of useful classification could be grouping of processes into a class of related processes, for example all children of a make command could be classified as a make-group and the scheduler can then apply a specific scheduling strategy for all processes in the make process tree.

The classification mechanism could be a sophisticated machine learning system. It does not need to run in a performance critical time, but instead it could analyze past execution traces offline, when there is a spare computing power available. Nonetheless, our current system is very basic and does use only a combination of statically defined regular expression rules for classification of process groups and process types (long running vs short running) plus it does a specific time prediction analysis for a gcc command based on a name of a file being compiled. Scheduler currently uses only the long versus short running time classification, to decide whether a preemptive migration is useful for a particular process. Introduction of more sophisticated classifications and better scheduling decisions based on those is a future work, but all required framework for this work is already in place.

3.8 Process trace dumper

When running applications in a cluster, they often do not perform as good as hoped. Therefore any clustering system has to provide some tools to debug cluster specific execution problems. Our system has various hooks in a kernel and nodes expose certain information about themselves to other nodes so we can provide various useful statistics.

An example of such a troubleshooting tool our system provides is a support for generating an execution trace of a distributed process. The motivation for tool is to help better understand relation between processes of some complex multi process application, such as compilation of programs with gcc and make. The execution dump shows relation between all processes executed (i.e. child to parent relationship), a node where a particular process has been executed, and as well a duration of execution. Based on our experience, this tools helps to understand which parts of a calculation are good candidates for migration and as well to identify sequential non parallelized parts or the program.
Chapter 4
Implementation

To verify our proposed concept in practice, we have implemented all key components of the ONDC architecture. We have based our implementation on the Clondike system [42, 83], which was initially developed as a non-dedicated clustering solution. It was a suitable starting point as non-dedicated clusters share a lot of common requirements with ONDC systems.

In the following sections, we discuss the most important areas of implementation. We focus primarily on discussion of higher layer features, like scheduling and security, and do not go much into technical details of migration mechanisms and all kernel implementation. The migration mechanism implementation itself is a very large topic and most of the details of our implementations could be found in [83].

4.1 Overview

Linux operating system is working at 2 basic levels, user mode and kernel mode. When implementing an SSI cluster with transparent clustering, implementers always have to carefully consider where to put each piece of the functionality. In order to achieve transparency and good performance, some features have to be implemented in a kernel mode. On the other hand, a kernel mode programming is complex, hard to debug, and errors in the kernel code could easily lead to corruption of a system.

In Clondike we paid a special attention to keep the kernel code at a minimum required size. In order to even further increase maintainability of the system, we have separated the kernel code into 2 parts - a patch to the standard Linux kernel and a set of modules. The patch needs to be modified with every new release of the kernel and so it has the highest maintenance cost of all code. To keep the patch size minimal, there is nearly no functionality, it rather provides only hooks to kernel where the modules can plug themselves to the kernel and implement the required behavior. As of now the patch is around 50KB large, so its maintenance costs are relatively low.

The second part of the kernel code are kernel modules. They provide all required functionality for transparent process migration, like process checkpointing and restart-
ing (Section 4.4), remote system calls, remote pipes (Section 4.6), distributed filesystem caching, etc. The kernel modules provide interfaces to user mode based on the pseudo filesystem \cite{18} and netlink \cite{37}. The kernel part is completely isolated from the user mode part by those interfaces.

The user mode layer is the easiest to program and debug and so most of the high level cluster functionality should be implemented in this layer. It is responsible for all high level cluster features, like scheduling, security, discovery, and health monitoring. The user mode layer could be developed mostly independently of the kernel part. We describe our implementation of the user mode layer, called director, in the following sections, but it is easy to replace the implementation with any other that conforms to the API exposed by the kernel layers.

![Diagram of high level layers of implementation.](image)

Figure 4.1: High level layers of implementation.

4.2 User mode director

In contrast to the kernel code, which has to be written in the C programming language, the user mode layer could be written in any programming language capable of using the kernel netlink API. In order to make integration easier, we have written a small library in the C language that wraps all netlink calls into simple C function calls. This library could be used to shield user mode implementations from tedious netlink communication details. Due to the C implementation, nearly any programming language can use the library, as binding for C libraries exist in nearly all languages. A possible language for implementing all user mode layer would be the C language, but since the user mode layer performance is not as important as ease of development and experimenting (at least at the current state of project), a higher level language is a more suitable choice.
We have chosen for our user mode code implementation the Ruby programming language. It is a dynamic strongly typed programming language popular for rapid application prototyping. It is not the best language from a performance point of view, but for the purpose of experimenting with various implementations of cluster features, it is a good fit. Ruby does not support on itself netlink interaction, but it supports C libraries binding via custom modules, so we have implemented a custom Ruby module library giving access to our netlink library from the Ruby language. We call the user mode application director, as it controls all high level cluster functionality.

Figure 4.2: Basic view of director components.

Figure 4.2 shows a high level decomposition of the director to its basic components. The Kernel API provides access to the kernel layers, while the interconnection is used to communicate with other nodes. The local API provides an access over local sockets to external applications, currently used only by a security module and a measurement framework. The security module is consulted for all access request evaluations and is as well responsible for distribution and storage of certificates. The scheduler with the task repository makes all preemptive and non-preemptive migration decisions. The membership component keeps track of connected nodes and their status. It is responsible for connecting to new nodes or disconnecting from currently connected nodes. The node repository holds known data about remote nodes.
4.3 Communication channels

The system contains several internal and external communication channels. Figure 4.3 illustrates the main channels used in the system. Following is a brief description of the channels roles:

**Netlink** Netlink channels are used for communication between kernel and user mode layers. All communication requiring exchange of any complex/structured data is done using this channel. Both kernel and user mode layers can initiate a message exchange. Any sent message does always generate a response or at least an acknowledgement message.

**Pseudo filesystem** This is an original kernel to user mode interaction layer that was present in the system before introduction of the netlink layer. It provides some basic data about system state and also offers several writable files to control the system. Initiation of a connection to a remote node and preemptive migration are examples of actions managed by this interface. The write part of the interface could eventually be replaced by the netlink channel, but read part should remain, as it provides a very easy access to some system information like connected nodes even when no director is present.

**Kernel** A core channel established between a pair of nodes when they get connected. The channel is used for all migration related communication, system call/signal forwarding, and node connection/disconnection. It is currently using the TCP protocol, but the system abstracts those channels away and it could be replaced by other protocols.
In reality, there are multiple kernel channels between any pair of nodes, but description of them is beyond the scope of the paper. Details of all the kernel channels could be found in [83].

**Director** This type of communication is used to exchange short messages between directors running on nodes in a shared network segment. It is implemented by sending UDP datagrams and it supports both broadcasting of messages and directed sending of messages to specific nodes. This channel is the only way how nodes could talk to each other before they establish kernel channel between them. All messages send on this channel are signed by their sender.

**Proxy** Proxy system channels serve proxy filesystem in exchanging pipe data, more details about proxy filesystem are in Section 4.6. Similarly as the kernel channels, these channels are also established between a pair of kernels.

**Filesystem** A channel used by distributed filesystem layer. In case of Plan 9 this is an unencrypted TCP channel. NFS can run as well over UDP channels.

**Local** This is a special user mode only channel that is used to connect user mode components to a director. It does expose an access to a director security module and as well some configurations. An example of a user of this channel is a distributed filesystem server that needs to consult security system for access checks. The channel is implemented using a local Unix socket.

We call 2 nodes **connected** if they have established the kernel channel between them. Establishing this channel leads either to opening of all other channels between the 2 nodes, or if the establishment fails, then the nodes are disconnected and the kernel channel is closed. The connection is asymmetric, i.e., one node always acts as a server (a core node) and the other as a client (a detached node). In ONDC, both nodes can act as a server and client at the same time, but those connections are considered independent and on Figure 4.3 we just capture important channels from a point of view of one interconnected node pair.

When 2 nodes are connected, a virtual channel between their directors is established. The virtual channel is formed by using the directors’ netlink channels and the kernel channels. So a director can send a message using identifier of a remote node directly to its netlink channel, this message is forwarded by the kernel channel and then shipped by a netlink channel on a peer node to the associated director. This channel can be used for direct message exchange between nodes and more importantly for status checking, see Section 4.5.

### 4.4 Migration mechanisms

Before a process could be migrated between 2 nodes, these 2 nodes have to establish a connection between them. The connection establishment involves opening of all required
channels as described in Section 4.3, but also preparation of special logical entities called migration managers. For each connection between a core node and a detached node there is a corresponding pair of migration managers; one on a core node managing the connection with a detached node and one on the detached node managing the connection with the core node. The responsibility of migration managers is to keep track of migrated processed, handle individual substeps of migration as described later, dispatch properly remote system calls, and signal forwarding.

Migration of processes is performed by the kernel level code. A non-preemptive migration could be initiated on execution of the `exec` system call (see Figure 4.4), while the preemptive migration could be initiated by sending a custom signal to a running process (see Figure 4.5).

Figure 4.4: The non-preemptive migration can be started only when a process invokes the `exec` system call. For each eligible process, the `exec` syscall sends a netlink message to the director to check if the process should be executed locally or migrated away. The scheduler is consulted and replies back using the netlink channel either saying the process should stay on the core node, or a node where to migrate the newly executed process. If the scheduler decides that the process should migrate, the process will not execute the `exec` system call locally, but instead it will initiate a non-preemptive process migration.

After the migration is initiated, the process is stopped and all its internal state is stored into a file. The captured state includes information like open files and pipes, all CPU registers, allocated memory, status of signals, etc. In case of the non-preemptive migration, the memory content does not need to be a part of a stored data, because the `exec` system call erases the process memory and a completely new memory will be created after `exec` is executed. This makes the non-preemptive migration somewhat faster than the preemptive migration, especially for programs with a lot of allocated memory. Figure 4.6 captures a basic flow of events after the migration process is initiated.
The stored file is called a process **checkpoint** and the process of file creation is call **checkpointing**. Checkpoints are always created prior a migration, either by a core node when migrating to some detached node, or by a detached node when migrating back to a home node. The checkpoints are stored on a core node filesystem. When they are made by a detached node, the checkpoint file is created using a remote filesystem.

Checkpoints contain all the information required to restore a process. It can be restored again on a core node or on a detached node. The restoration process works differently on a core node and detached nodes. On detached nodes, first an ordinary process is created using the `fork` system call. The newly created process is called a **guest** process. After the process creation a distributed filesystem of a core node is mounted on a detached node, and then the newly created guest process root directory is changed to the root of distributed filesystem\(^1\), so the migrated process shares the same filesystem view as the original process on the core node. After the filesystem is ready, the new process invokes the `exec` system call on the checkpoint file to be restored. A custom handler for checkpoint files is initiated and it restores the process state based on the checkpoint data. As a part of a process state restoration, all open files of the process are reopened through distributed filesystem and also, its pipes are open using our special purpose filesystem, see Section 4.6.

When migrating back to a core node, no filesystem mounting and root directory changing is required. In addition, no process forking is required, because the process on the core node is restored to the original process instance where the process was running before being migrated. The original process instance is never destroyed until the process termi-

\(^{1}\)Using the `chroot` system call.
Figure 4.6: After the migration process is initiated, the flow of events is mostly the same for the preemptive and non-preemptive migration. The current application is frozen and a checkpoint is made. After the checkpoint creation is completed, the kernel informs a peer node kernel that the checkpoint is ready. The peers node kernel consults its director, whether it is willing to accept the process or not. If the director rejects the migration, the core node is notified back and the migration is aborted and the process runs locally. Otherwise, the migration proceeds.

nates and it is called a shadow process. The shadow process exists also when a process is migrated away to a remote node, because it holds a residual state of the process on its home node. The residual state is represented by all environment dependencies that cannot be easily migrated away, like relation with parents and children, or open pipes and sockets. All system calls that need some data on a local node are forwarded by the guest process to the shadow process and the shadow process executes them and forwards results back to the guest process. Examples of such system calls are process coordination calls like wait, or process hierarchy related calls like getpid, or getppid.

Thanks to the checkpoint-based migration mechanism, the system supports also limited rollback and recovery mechanisms. For some processes that do not have any external dependencies like open sockets or interactive pipes, the system can just take their check-points and restart them without actually doing any migration. If the restart was initiated due to a remote failure of a process, it is called a recovery. If the reason for the restart was a detection of security violation on a remote node, it is called a rollback. Those mechanisms are not applicable for a broad range of tasks, but a lot of scientific calculations performing intensive computations with a very limited environment interaction can be successfully recovered.

The process migration is a technically complex topic and we have only briefly touched the main areas. Details of our process migration implementation can be found in [83] or
directly in the publicly available code repository [21].

Currently, our implementation supports most of the standard system calls including support for pipes, signals or process forking mechanisms. The main missing functionality is a support for migration of multi-threaded applications. We have an experimental implementation evaluated [61], however it has not yet been merged into our main code.

4.5 Monitoring and information dissemination

Each node participating in a cluster periodically propagates static and dynamic information about itself. The static information includes CPU model information, available memory, node identifier, and its address. The most important dynamic information is the current load, CPU usage, the number of locally running cluster tasks, and a willingness to accept more jobs. The information is disseminated using a UDP channel by performing a message broadcast. For larger networks spanning over multiple networks segments a different technique needs to be used. Since the freshness of the information should not and is not crucial for any of used cluster algorithm, any gossip-based techniques can be used to slowly disseminate information through the cluster [32].

The static information is used by nodes to keep track of other existing nodes in case they need to connect to more detached nodes. The dynamic information can be used in scheduling, but due to lack of guarantees about freshness of this information it is useful mainly for monitoring purposes.

A second channel used for node monitoring is a virtual channel formed by the netlink and kernel channels as described in Section 4.3. This channel exists between any pair of 2 nodes that are connected. Both nodes periodically send a heartbeat message over this channel and their peer monitors that those messages properly arrive. If a certain amount of heartbeat messages is missed, a node considers its peer node dead and terminates all connections and removes the node from a list of active peers.

An advantage of using this type of channel for monitoring is in its ability to detect problems in multiple layers. If only pure user mode channels are used, a blocking problem in kernel layer could remain undetected and a blocked peer could be considered alive. In addition, using this virtual channel makes the monitoring oblivious of network specifics; a director just sends the message to its local netlink channel and waits for a response. If a direct connection is used, the monitoring framework needs to know how to get packets to its peers.

4.6 Proxy filesystem

Linux uses heavily a concept of pipes [18] both for inter process communication, and for standard input/output passing. Without support for migration of processes with pipes,

---

2This is easy in simple networks, but may be more complicated in larger networks with many address translation layers.
possible use cases of our system would become very limited. Our system does support pipes migration through a pseudo filesystem we call the proxy filesystem. The basic idea of the solution is that whenever a process is migrated for the first time from its home node, its shadow task creates a wrapper around existing pipes that will reside in a so called proxy filesystem server and will not be closed until a process requests closing of associated pipes. A guest part of the process running on a detached node will use a special privately mounted proxy filesystem. This filesystem will contain corresponding proxy files that will transparently communicate pipe traffic with the proxy server on the core node. If a process migrates back to the core node, the proxy file representation remains, so the process will mount on a core node a private copy of the proxy filesystem and communicate with a (local) proxy file server through this file. It is out of the scope of this paper to provide details about the proxy filesystem, they can be found in [60].

4.7 Scheduling

The scheduler component is responsible for all migration decisions. There are 3 types of migrations, but only 2 of them are directly initiated by the scheduler. The following is a brief description of 3 possible migration scenarios.

Non-preemptive Non-preemptive migration can be initiated by the scheduler when the exec system call is executed. The kernel patch adds a hook into the beginning of the exec call, the kernel modules plug into that hook and forward all information about exec to the user mode using the netlink channel. The scheduler is listening on those events from the netlink channel and it responds to the kernel using a netlink response either specifying a node where a new process should migrate, or telling the kernel to proceed with local exec with no migration. Figure 4.4 illustrates the event flow of a non-preemptive migration.

Preemptive migration from a core node The preemptive migration can be initiated at any time. The preemptive migration is initiated by writing a process identifier and an identifier of a target node into the pseudo filesystem channel. This in turns sends a special signal to the process and migration is initiated. The scheduler is running a background thread that periodically checks whether a preemptive migration should be initiated and if so it informs the kernel layer through the pseudo filesystem. Figure 4.5 illustrates the basic flow of events in case of the preemptive migration.

Preemptive migration from a detached node This type of migration is not directly initiated by the scheduler, but rather by a detached node. It is initiated when a local user start interacting with computer and so the system wants to free up resources for a local user. This is not performed immediately, but rather with a short delay to give short running processes some time to finish. The core node scheduler is as well involved in this type of migration as it can decide a new node where a task could be
The scheduler itself is implemented as a generic (ruby) object listening on `exec` commands and checking for load balance in a background thread. The scheduler object does not perform any decisions on its own, but instead delegates those decisions using a strategy pattern \[31\] to specialized objects. Implementing a migration strategy can be as easy as writing a single ruby object with at least 2 methods required by the scheduler, one for non-preemptive migration decisions, second for rebalancing decisions used by preemptive migrations. We have implemented a couple of strategies for testing, they include:

**Round-robin** The scheduler assigns tasks to all known nodes in a circular fashion.

**Random** Targets of migration are chosen randomly.

**CPU-based** A target node with the lowest CPU load is chosen.

**Quantity-based** The number of tasks running on a remote node is tracked and the scheduler tries to balance this number.

The strategies could be easily combined. For example, it would be easy to implement a strategy containing all others and switching between them based on a program being executed. The round robin and random strategy could work well for a large list of tasks with short runtimes. The CPU-based scheduling is generally not very useful. First, for short duration tasks the CPU usage can be changing really fast depending on a current number of tasks and the information on a core node is always a bit outdated. For long running tasks, making scheduling decisions based on a current or recent CPU usage is not very practical, as the task may be running for many hours or days and so a current CPU usage is likely not very indicative.

Based on our observations, from the strategies above, the most useful is the quantity-based strategy and this is also a strategy we have used in all measurements presented in this paper. The strategy has been described in Section 3.5.1. An important component for this strategy is the task repository. This is a component contained in the scheduler and it is used to keep track of all cluster tasks running locally and remotely. It is possible to construct this list, because task repository listens on hooks of `exec`, `fork`, and `exit` system calls, and as well on all results of migrations. This way it knows where all the tasks are running. The hooks for the `fork` and `exit` system calls work similarly as the hook for the `exec` call described earlier.

In addition to keeping track of processes and their current locations, the task repository also tracks a history of all nodes visited during process execution and operations they have performed. This information is important for stigmata mechanism when making security checks for a particular process. Forward stigmata information could as well be kept in the task repository. The forward stigmata can be used by the scheduler to prevent mistakes.

---

\(^3\)Currently, we do not support direct migration between 2 detached nodes, so a task must get first to a core node and only then can the task be migrated to other detached node.
and never migrate a process again to the same node where some security permission was violated. Ordinary stigmata are available to the security module to evaluate access checks. As a future extension, prediction and classification mechanisms (see Section 3.7) could be used to predict what files will a process require, and based on this, the scheduler could try to avoid migrations that will lead to violations.

4.8 Security

A key mechanism for protecting detached nodes from remote processes running on them is based on limiting access privileges of those processes. They are executed with permissions of a special low privileged user that does not have any special permissions on the detached node. The only time when a process needs some special permissions on a detached node is when it is being initiated, as it needs to perform privileged operations like mounting a remote filesystem or changing the root folder. The initial guest process setup is done with higher privileges, but the setup itself does not execute any code specific to the program, it is only the cluster system code that can be assumed trusted. After the setup is finished, the remote program itself is executed, but it does not need any special privileges. All privileged operations are performed using system calls and those are intercepted and either propagated to a core node for evaluation, or evaluated locally in a special way. Most of the calls, like signal sending or socket opening, are sent to a core node for security checks against a real process owner identity. The most significant exception to this rule is filesystem access that is performed directly and so, the filesystem layer needs to grant access to the special low privileged user being used by the cluster. Some other system calls are performed locally without any security checks, for example calls like \texttt{getpid} or \texttt{getppid}.

Stigmata, proposed in [43], are a mechanism to protect core nodes against malicious modifications of running programs made by detached node owners. The main part of the stigmata mechanism is implemented as a module of the director component. This module has 3 main responsibilities:

1. Manage certificates. This includes both managing the certificate storage and controlling distribution of certificates.

2. Evaluate security requests whether they match all stigmata rules as described in Section 3.4.3.

3. Provide an interface to other components and external processes, to be able to query the security evaluation system.

The rules what certificates are stored on a node were described in Section 3.4.2. The security module obeys those rules and keeps required certificates stored locally on a core node hard drive. For performance reasons, all rules represented by certificates are kept in memory. The privileges granted to individual entities are kept in a hash-based lookup table that allows $O(1)$ lookup. The rules are then all sequentially evaluated until a matching
rule is found. This works well for cases with at most a few hundreds rules per each remote entity, which we consider a typical case. A simple future optimization could organize the rules based on their types, i.e., separately rules for a filesystem access, for signal sending, etc. This would reduce the number of rules that need to be checked. A more sophisticated approach would be usage of a pattern matching system like that based on the Rete algorithm [28].

By keeping rules in memory, both lookup and parsing times are saved on rule evaluation time. Based on our calculations, in a cluster of size of about 1000 nodes the required memory would be around 5MB, which is a negligible amount of memory in today’s workstations. If the memory consumption proves to be a problem in future larger clusters, it is possible to optimize memory usage by keeping in memory only rules affecting currently connected nodes.

The certificates are represented as plain text files. For security features like hash computation or message signing, the OpenSSL [69] library was used. Asymmetric cryptography features rely on the RSA-based cryptography [75].

The security module provides 2 ways for other modules and applications to interact with it. First, the kernel layer can send evaluation requests through the netlink channel and the director will dispatch those requests for evaluation to the security module. This is useful for checks initiated from the kernel, like sending of a signal. The second type of checks is done through a special library that could be linked to user mode programs, which can use it to request a check. The library in turn forwards this request via a local Unix socket to a director process and it will evaluate it and respond. This type of interaction is required for user mode processes that require interaction with the security module, for example, a filesystem server. Figures 4.7 and 4.8 illustrate 2 ways how a security check could be performed when the check is initiated locally and Figure 4.9 shows a flow of events in case a security check is initiated due to a remote system call.

4.9 Filesystem integration

There are 2 main aspects of the filesystem integration. First, the system must be able to automatically mount a filesystem on remote locations and use it as a root directory for migrated processes. Second, the system must integrate with the cluster security system.

The automatic mounting of a filesystem is handled by kernel modules. Whenever a new process is migrated to a detached node, the module tries to mount a filesystem for it. The filesystem is mounted into a temporary directory and if the mounting succeeds, the process can then change its root into that directory. In order to enable possible caching benefits of a custom sharing layer, the mounting must not be done on per process basis, but rather on per node basis. That is, the filesystem is mounted into a temporary directory only first time some process comes from a particular remote node and all other processes

---

4Some filesystem servers can reside directly in the kernel, like the one for NFS. Then the netlink channel could be used. However, other filesystems, like Plan 9, have only user mode servers.
coming from that node will reuse this mounted directory.

To integrate security, both a kernel level filesystem running on detached nodes and a server running on core nodes needs to be modified. The changes to kernel modules of a filesystem are required in order to preserve access to user owned files. As mentioned in Section 4.8, migrated processes run with access permissions of a low privilege user. If the filesystem layer would present a real id of file owners, the user can be denied access to his files. To illustrate the problem, let us assume that the virtual user id on detached node is 999 and a real user id on home node is 20. In addition, we assume that the access privileges to file are configured so that only the owner can read it. If the remote filesystem presents the file as belonging to user 20, a process running with privileges of a user with id 999 would not be able to access the file. In order to address this problem, the filesystem layer has to present the files belonging to the user associated with the virtual user as belonging to this virtual user id, i.e., in our case to the user with id 999. The same id translation mechanism should be performed for groups.

In addition to the user id translation, the filesystem layer should provide an identifier of a detached node to the filesystem server running on the core node. The filesystem server needs to be modified, so it accepts the detached node identifier. Whenever a file is being open, the filesystem has to send a process identifier of a process performing the operation and the filesystem server is responsible for checking the operation with the security system, as illustrated on Figure 4.10. For that, the provided API and a native library could be used as described in Section 4.8.

\footnote{In practice, this is a bit more complicated, as due to security as well a remote user id has to be considered, so there is one mount point for each combination of a remote node and a user ID from that node.}
Figure 4.8: Flow of events in case a security check is initiated directly by some user mode application. Typical examples of this use case are security checks performed by an user mode filesystem server. Responses use the same channels in the opposite direction.

4.10 Caches

As mentioned in Chapter 3, a long term goal is to have a suitable cache coherent distributed filesystem, but currently we rely on our own simple filesystem caching layer. This caching layer is implemented as an overlay Linux filesystem. It can be mounted over an existing filesystem and it automatically adds caching capabilities to selected files. The caching layer is implemented as a kernel module. The module implements all essential structures related to filesystems, like inode, dentry or address space. Most of the calls are just forward operations to underlying real filesystem structures, but a few calls were modified to enable standard Linux page-based caching on address space operations. The most important part in enabling the caching is marking dentries as cacheable and implementing properly associated address space operations. For more details about how Linux filesystems work, see [18].

The overlay filesystem can be mounted at any possible mount point (directory) and it will cache all files and directories in a tree starting from the mount point. In addition, the mounting supports 2 special caching modes that control what gets cached. The first simple mode just does not cache files created or modified in the last hour. The motivation for this strategy is in compilation of files, where some transient files are automatically generated and they are not suitable objects for caching, since they may change frequently. The second mode allows a user to specify a pattern of filesystem names that can be cached. For example, a user can specify that only *.h files (C/C++ header files) are cached.

The caching filesystem is automatically mounted by the director when it detects that a new remote filesystem has been mounted. The director detects this by getting a netlink
Userspace layer

3. Netlink

Userspace layer

2. Syscall forwarding

Kernel layer

Director

4. internal

Security

Kernel layer

Application

1. Syscall

Figure 4.9: Flow of events in case a security check is initiated due to a syscall invoked on a remote node. Responses use the same channels in the opposite direction.

message from a kernel informing about a newly migrated remote process. The caching filesystem is now automatically mounted to a few well known locations like /usr/bin, /usr/lib or /lib. For higher flexibility this configuration should be exposed to users.

An impact of caching on security, most notably stigmata has been discussed in Section 3.4.4. Except for the problem with enforcing stigmata, there is also a technical problem with tracking of file open operations. A core node has to track all open files to be able to properly evaluate stigmata rules. However, if there are caches, open calls may not be propagated to the core node. It is not sufficient to detect this information when a process migrates back to a home node, because the file may be already closed before that, but it is still a relevant stigmata information. The problem can however be addressed by sending asynchronous open notification from detached nodes to core nodes on file open operations. If a malicious owner of detached node decides to violate the protocol and ignore sending of those notifications (by altering a cluster code), it does not bring any new security threat, because the owner of the machine cannot get access to any more new files than he was granted by the core node owner.

4.11 Mixing 32 bit and 64 bit computers

Clondike was originally implemented for 32 bit systems only. These days a majority of computers is 64 bits and so the Clondike code was ported to support both 64 bits and 32 bit Linux operating systems. Since currently many systems running 32 and 64 bit operating systems coexist, it is a natural question whether a cluster can use both at the same time. On Linux, it is possible to run both 32 bit and 64 bit programs on 64 bit operating system,
but only 32 bit programs could be run on a 32 bit version of the operating system.

We have implemented a proof of concept for support for mixed clusters. The required steps to achieve this are:

1. All messages exchanged between clusters have to be machine bit width independent, i.e. usage of native types like int has to be avoided, and instead a fixed length data types like int32, uint64 have to be used.

2. When a 32 bit process from 64 bit machine is migrated a fixup work is needed on stack as syscalls are performed differently on 32 bit and 64 operating systems.

3. When a 32 bit binary is migrated to a 64 machine, stack again needs to be modified to follow 64 system call conventions. In addition a VSDO page has to be mapped to a proper place of 32 bit process.

4. Thread local storage information and associated gs/fs registers are handled differently on 64 bit and 32 operating systems and a it needs to be converted if migrating between platforms.

5. Scheduler needs to be aware of differences between 32 and 64 bit machines and never attempt to migrate 64 bit binary to a machine with 32 bit operating system.

---

6 This is normally done when executing elf binary, but if the binary was originally executed on 32 bit system it was not required and so we need to manually add it.

7 Our scheduler is currently not doing this, but it is an easy addition.
Chapter 5
Experimental Evaluation

ONDC-based clusters can be used for many different purposes and it is out of the scope of any single thesis to provide detailed tests of all possible scenarios. As a demonstration of system capabilities, we have chosen some example types of calculations that could be performed by a cluster and demonstrate performance characteristics on them.

We provide detailed information about our laboratory setup and measurements that were performed. However, the data could still differ in different network setups, we have for example observed some asymmetric behaviors in the network, where calculations started from some computers took (consistently) longer time then from others, even though all machines have exactly the same configuration.

Our system is publicly available and anybody can download it and experiment with it [21]. In addition to the source code, also definitions of most of the measurements performed in this work are available as a part of our public repository.

5.1 Measurements framework

Unlike as in traditional clusters, in ONDC any node can start a new processes to be run in its own cluster. This makes it more difficult to perform measurements, as they cannot be always simply started from a single machine. In order to simulate multiple workstation users, it is sometimes required to initiate calculations from multiple different workstations. In order to make testing process easy and repeatable, we have developed a simple testing framework for measurements. The framework allows definition of a measurement specification in a single file and then execute the measurement from a single node. The definition contains a number of nodes to participate in the measurement and tasks that should be run by each of the node. Each of the tasks has associated a start time\(^1\).

The framework runs on top of the user mode communication layer using the director channels. The initiating node parses the measurement plan definition and dispatches it to all known nodes. If there are more nodes than the number of required nodes, some of the nodes get a special suspend command so that they do not participate and they are resumed

\(^1\)Defined as an offset to current time.
only after execution is finished. Each of the nodes is assigned a single part of the plan and it takes care of it execution. We provide 2 special predefined tasks that could be run: a busy-block and sleep-block tasks. They serve as a simulation of local user activity, either with high CPU usage (busy-block), or with no CPU usage (sleep-block). Both of them cause a node that executes them to reject incoming cluster tasks and emigrate back home any remote tasks running on that node.

An example of a measurement plan could be:

**Nodes count:**
12

**Bind:**
- SecondMaster: 192.168.45.135

**Tasks:**
- makekernel: exec "make -j 60 vmlinux" /mnt/ext/linux-2.6.32.5
- ls: exec "ls -la" /tmp

**Nodes:**
- LocalNode: makekernel
- SecondMaster: makekernel 20
- RemoteNode1: ls 40

The first section specifies how many nodes participate in the measurement. The Bind section is optional, if some of the nodes has to be bound to a specific machine. It is not required, the nodes could be assigned randomly, but for deterministic repeatable results it is often better to bound key nodes to specific machines. The Tasks section defines what task should be executed; in this case we define a compilation of a Linux kernel and simple directory listing task. The last section tells which node should execute what tasks and when. The LocalNode is an alias for the node where the plan execution is initiated. Other nodes are named RemoteNode1, RemoteNode2, etc., or by a specific name declared in the Bind section. The node name is followed by a name of a task to execute on that node and a time offset when the execution should start. So, for example, the SecondMaster node will execute task defined as makekernel, but only 20 seconds after the plan was initiated.

It is important to realize that the measurements frameworks does only decide which nodes start which tasks, but not where the tasks will end up running. Those tasks can be migratable cluster tasks, so they may end up running on other nodes participating in a cluster. So for example, if a plan says RemoteNode3: makekernel 30, it only means, the node that has been chosen to act as RemoteNode3 will 30 seconds after the measurement starts execute locally the command corresponding to makekernel. This command will in turn execute a lot of compilation subtasks, and those subtasks could be migrated to any eligible remote node, including the LocalNode from which the plan execution was initiated. The measurements framework itself does not have anything to do with a real clustering
functionality and in a real production environment it should be disabled. It is used only to avoid need of manually executing tasks on many different nodes.

During the execution, the framework gathers key performance metrics from all participating nodes and provides them as a part of execution result log. The collected data include CPU usage, load, and the number of local/remote tasks being executed on each node.

5.2 Measurements environment

The measurements were performed in a university computer laboratory, using up to 14 identical nodes. The nodes were equipped with a dual core Intel Pentium(R) CPU model G6950 running on 2.80GHz. Each node had 4 GB of memory available. Every machine was connected to a shared switch via its 1 Gbps Ethernet network card. All machines were running 64 bit version of Ubuntu operation system with custom Clonkike kernel and code.

No channel security in form of IPSec was used in tests to secure traffic. We generally expect IPSec to be used in case a non-local network is used, so no channel security seemed to be a realistic setup.

Our tests are not bound to the environment we have used. Thanks to the public availability of our code and test definitions at [21], anybody can test the performance in any other environment (including heterogeneous clusters), or on any other type of parallel application supported by our system.

5.3 Execution measurements in homogeneous cluster

Each of the measurements has been performed 3 times and we present the best measured results of each of the measurements. We run all experiments with caches enabled. Initially, after computers are booted, the caches do not contain any data, we say they are cold. After a single execution all participating nodes get all binaries and dependent libraries into their caches, we call such caches hot. The hot caches contain as well some other data used by programs, but binaries and libraries have major impact on the results. All presented results are with hot caches. With cold caches, the slowdown depends on the total duration of the computation and also on the number of times the binary is executed remotely. In our longer experiments, a typical slowdown was in order of 10-20%, but this number could vary a lot depending on conditions. It is still much better than it would be in case of no caches, our basic experiments indicate typically an order of hundreds percents slowdown.

We have performed 2 main sets of experiments in a homogeneous cluster setup. First set of experiments is a parallelized Mandelbrot set calculation, the second is a compilation of a Linux kernel. We discuss each of these separately in next sections.
5.3.1 Mandelbrot set calculation

This type of computation represents one of the easiest type of cluster calculations, falling into a bag of tasks category of computations. The parallelization of tasks is performed by splitting an area of computation into equally sized portions and calculating each of them in a separate subprocess\footnote{Created using standard fork mechanism.}. The subprocesses write their results into their own files and those files are merged in the end of calculation. A slight complication is that equal size portions do not correspond to equally demanding calculations, contrary there are huge differences in processing times of each portions. So it would not be an optimal strategy to split task to exactly the same number of subtasks as we have available computation nodes, as some nodes would finish much faster than others and we would have to wait for the slowest node. A better approach for such a calculation is to split the task into a larger number of subtasks than the number of available nodes and each of the nodes takes several subtasks. The inequalities will still exist, but will be typically lower and it is also possible to better use rebalancing by preemptive migration.

The measurements presented here have generally very short running time, never longer than 2 minutes. This is not a realistic time, in a real world the duration for demanding computations would be much larger, likely in order of hours or even days. We choose this short duration only for practical feasibility of performing a sufficient number of measurements. The cluster overheads are larger in shorter computations as larger part of computations is spent by initial process and data distribution, so the presented results can be considered as the worst case efficiency of what could be achieved.

Local run with an increasing number of subtasks The first presented measurements are only from a local run on a core node with no detached node. They illustrate an impact of the number of subtasks on the total duration. It is clear that we need to use at least 2 subtasks even locally, because all our machines have 2 CPU cores. Similarly, in a cluster we need to use at least 2 times the number of nodes. However, as Graph 5.1 shows, we need a larger number of tasks to achieve optimal numbers. This is due to the inequalities of computational complexity of individual subtasks as discussed earlier. As the graph suggests, we can get some speedup up to around 30 subtasks. We use 100 subtasks for all clustered measurements, there is no important slowdown and with 30 tasks only we may again get into an imbalance problems on a full cluster (where we have 28 computation cores available).

Single computation A first measurement involving task migrations is a single Mandelbrot set calculation task run on an increasing number of nodes. To get an idea about scalability, we compare the observed times with a theoretical ideal time. The theoretical ideal time is simply a time, that could be achieved if there are no overheads of using cluster. In this case, it could be calculated as a duration of calculation on single machine, divided by the number of participating nodes. Such a time cannot ever be achieved in
5.3. EXECUTION MEASUREMENTS IN HOMOGENEOUS CLUSTER

As Graph 5.2 illustrates, the clustering is scaling well, the cluster keeps a relatively stable time overhead compared to the ideal time. In absolute numbers, the overhead is between 5-10 seconds, in percents numbers it is growing from around 25% to over 100%. The increase in percents overhead is expected, as the ideal time of calculation with 14 nodes is very short, it is around 6 seconds, which is a low number in clustering terms as just a migration can take a few seconds. Since the tasks do not make any remote operations except for migration in startup and then remote waiting on the results in the end, we can consider the absolute overhead in seconds to be more representative and this is the number that would be achieved in longer runs.

The graph also indicates some problem when measured with only 2 nodes. This case was always special and in all measurements it suffered a very large disproportional slowdown. The reason for problems in this case is most likely due to our scheduling algorithm that prefers to migrate away as much tasks as it can and keep locally only a small number of tasks. This is generally a good strategy, but with very short tasks and only a small number of remote nodes it can lead to highly suboptimal numbers.

**Single computation with interruption**  This measurement was performed to illustrate preemptive migration capabilities and minimal interference requirement satisfaction. The measurement is run on 1 core node and 5 detached nodes. 5 seconds after computation starts, 2 of detached nodes start locally a simulated user activity task taking 100% of one
In order to satisfy the minimal interference requirement, the cluster should stop using the nodes running locally demanding tasks. This is done by initiating preemptive migrations on affected nodes and migrating remote tasks back. As discussed earlier, this is not done immediately, but rather with a some delay (10 seconds in our measurements). The Graph 5.3 captures the number of cluster tasks running on each of the nodes. It can be observed, that after 15 seconds the remote tasks are being migrated away from 2 affected nodes and the core node starts to dispatch those tasks to remaining detached nodes.

The Graph 5.2 illustrates a CPU usage of all nodes during the computation. The 2 detached nodes with local tasks end up with around 50% CPU usage, as they are running a simulated single thread local user task, so it keeps up busy one core out of 2 available. In theory, cluster could still use the second core, but currently we prefer to have a conservative scheduler that gives higher priority to minimal interference requirement, so that it does not drive users away from participating in the system. It can be also observed, that core node is mostly underutilized, its utilization spiking only when the remote nodes migrate tasks back and core node emigrates them again away. Based on our experimentation with system, it is generally safer strategy to keep core node underutilized than to try to fully utilize it, as when it becomes fully utilized, it may not have enough power to distribute cluster tasks to other nodes fast enough, resulting in overall computation slowdown.

In terms of total speed, the best duration of this measurement was 34 seconds, which is equal to best duration when we run cluster with 3 nodes. This experiment could run faster, as it could utilize 2 more nodes for first 15 seconds, but preemptive migration is
5.3. EXECUTION MEASUREMENTS IN HOMOGENEOUS CLUSTER

Figure 5.3: A graph capturing a number of cluster tasks running on each of cluster machines during distributed execution of a Mandelbrot set calculations with a simulated user activity on 2 nodes.

much more time consuming than non-preemptive due to required memory transfer, so the numbers can be considered good.

**Multiple computation** In the last measurement, we finally show a full power of ONDC architecture. In this measurement we let more and more nodes to play a role of a core node and use others as their detached nodes. This is a scenario, that cannot be performed with traditional dedicated or non-dedicated clustering solutions, because they do not support starting execution from all cluster nodes. It is important to realize, that even though all core nodes execute the same cluster computations, it could be easily the case they execute completely different calculations on completely different data. More importantly, the detached nodes do not need to know anything about the program being executed or its data, the program, all required system libraries and all data are automatically provided by the system to nodes that need it.

We performed the measurement on a cluster of size of 14 nodes. The same Mandelbrot set calculation as before was started on more and more core nodes. All nodes were acting as detached nodes, but due to the minimal interference requirement the nodes that were at the moment running their own Mandelbrot set calculations were not accepting any new remote tasks. They started accepting tasks only after they were done with their own calculation. The Graph 5.5 shows an average duration of all computations compared to the ideal duration. The average does not mean we make an average of individual runs.
Figure 5.4: A graph capturing a CPU usage of each of cluster machines during distributed execution of a Mandelbrot set calculations with a simulated user activity on 2 nodes.

of measurements, we again show the fastest result achieved. The average is made among multiple simultaneous runs, so if we have 3 simultaneous core nodes doing Mandelbrot set calculation, we present an average of those 3 calculations. The ideal time is calculated similarly as in single Mandelbrot set calculations, but in this case it must be multiplied by a number of simultaneous computations. The overhead in absolute numbers is again quite stable between 10 to 20 seconds. The relative overhead is decreasing with a number of simultaneous calculations.

5.3.2 Kernel compilation

The second set of measurements does not use any artificially created programs, but rather performs a real world computation - a compilation of a standard Linux kernel (version 2.6.32.5)\(^3\). The kernel configuration file used for the compilation can be found as well in our online public repository.

This is, indeed, a very important type of calculation as this is exactly where systems like ONDC could be useful. A compilation of a large software base is happening frequently in many existing academic and industrial environments. Since the compilation is demanding, typical machines for developers are high end machines with 4 or 6 CPU cores. A typical compilation usage pattern is that a user compiles once in a while and for the rest of the

\(^3\)To be precise, we do run only the “make vmlinux command”, and do not perform final bzImage creation. This part is not parallelized and hence is not interesting from our point of view.
5.3. EXECUTION MEASUREMENTS IN HOMOGENEOUS CLUSTER

Figure 5.5: A graph capturing a duration of an execution of an increasing number of Mandelbrot set computations running on a full cluster.

time he does not need all power of his machine. This is the time, when ONDC can help as it allows other users to use unused computing power (or just some cores) of other nodes. This is not possible to achieve with dedicated clusters, as the dedicated clusters cannot be build from workstations used by different users. Traditional non-dedicated clusters are also not an answer to this scenario, because only one node can act as a core node, while others only contribute their computing power.

All tools used during the measurement are original unmodified programs in the same form as they are used normally. The transparent clustering functionality is achieved thanks to the kernel level layers of our system. The only thing we do differently when running the compilation in cluster, we set manually a high parallelization (-j) flag to the make command. Similarly as in the Mandelbrot set calculation, we set a maximum number of subtasks to a fixed number, same in all computations. The number used was 60. In a future, setting of parallelization for compilation could be also done automatically by the system, but it is not supported now.

There are 2 major differences from the Mandelbrot set computations affecting possible scalability:

1. Not all subtasks are started in the beginning, but rather they are being continuously started during the compilation. In addition, the number of active tasks is not stable, but it varies with time.

2. Tasks perform a lot of interaction with filesystem. First a task needs to read the file
to be compiled, then all associated header files have to be read, and finally the task needs to write the assembly output back to the core node.

**Single run** In a basic measurement, we run a single compilation from one core node with more and more remote nodes. The graph 5.6 illustrates scalability of the solution. As the graph shows, the solution scales to about 6 nodes, but then no measurable speed ups were observed. There are couple of reasons for the limited scalability.

![Graph showing scalability](image)

**Figure 5.6:** A graph capturing a duration of a compilation using Clondike on an increasing number of nodes.

1. Inherent scalability limits of the system. The ideal duration on the graph is calculated the same way as for the Mandelbrot set computation. However, the situation is much more complicated here, as the tasks are not all created immediately and at some points there is much lower number of tasks than the required number of tasks. In addition, there are couple of inherently sequential parts, like generating of initial system dependent header files, or linking. Those also contribute to limited scalability.

2. Core node overloading. We observed this effect with larger counts of detached nodes. It is partly caused by suboptimal scheduling, when core node keeps locally more tasks than it should. The other part contributing to the core node overloading is a suboptimal implementation of the user mode director in a high level interpreted language, as it sometimes consumes up to 25% of CPU. A native implementation may help to reduce this number.
3. Limited distributed caching. We do cache only binaries and stable header and source files, but header files generated during the compilation are not cached. There is a couple of those generated header files used frequently and a repeated reading of those file puts a pressure on the core node, consuming also over 25% of its CPU capacity at peak load.

In order to better assess quality of our results, we have compared our results with a special purpose system dist-cc. This is a solution designed for parallelizing of compilations. It has some problems with a compilation of a Linux kernel as some of its optimizations cannot cope with the auto generated header files. Nonetheless, with careful configuration of the dist-cc system, it can be used to compile the same kernel (with the same configuration) as compiled by Clondike. Achieved times (see Figure 5.7) are a bit faster than ours, but the scalability limit is as well 6 nodes.

![Figure 5.7: A graph capturing a duration of a compilation using dist-cc on an increasing number of nodes.](image)

Unlike our solution, dist-cc is a very specialized application that can do only parallel compilation. It requires a static preconfiguration of used nodes and a specialized single purpose software to be installed on every machine. Compared to this specialized solution, results with our generic solution are encouraging.

**Two runs** The ONDC cluster is not limited to a single cluster computation at the same time. This is what makes the system really scalable, as with growing cluster size there can be more nodes wanting to use cluster computation power and so even a limited scalability of individual computations may not be a problem.
In this measurement, we execute 2 kernel computations from 2 different nodes at the same time. Similarly, as discussed in the multiple Mandelbrot set calculations, those do not need to be the same compilations. In fact, we could well run a kernel compilation on one node and a Mandelbrot set calculation on another, but the results would be more difficult to interpret. The source code used for compilation is independent, each of the core node starting the compilation must have its own copy of kernel source code. The versions being compiled could be different, but this would again make interpretation of results more difficult.

As Graph 5.8 shows, the solution scales roughly to double the number of nodes, but we can observe a small decrease of compilation duration even with addition of 14th node. This illustrates the fact that more core nodes can saturate larger clusters.

![Figure 5.8: A graph capturing a duration of a compilation of 2 kernels simultaneously on an increasing number of nodes.](image)

An important indicator of scalability is a ratio of detached nodes that could be effectively used per core node. For kernel compilation, we achieve a measurable speed benefits for about six nodes. This number is not very large, but when we compare it with a specialized solution like dist-cc, this can be considered a good result. As the measurement with 2 parallel compilations shows, the ratio of efficiently usable detached nodes to core nodes stays roughly the same with growing cluster size, which is a desired result.

### Multiple runs

In a next measurement we run an increasing number of simultaneous kernel compilation on a full cluster, i.e. on 14 nodes. Graph 5.9 captures results compared to theoretical ideal times. An ideal time is simply calculated as a duration of an execution
on a single node, multiplied by the number of simultaneous compilations, divided by 14 as a number of participating nodes. As the graph shows, the time of 1 up to 4 kernels is quite stable, indicating the cluster is still not fully utilized. With 4 kernels, we have 3.5 node per calculation and we achieve about 2.08 time speedup which is still close the best speedup we were able to achieve with current implementation.

The results indicate, that having one out of 4 users compiling at the moment can lead to roughly doubling of overall system performance compared to network of independent workstations. Having one out of 5-7 users doing a computation leads to more than 2.5 times speedup. It is worth noting, that no special configuration what task goes where is performed, everything happens automatically. Due to decentralization of the system, the scalability is not limited by the size, the main limiting factor in our case would be a switch connecting all nodes together. In a larger network, the network topology is more complex and only parts of the networks share the same switching device. In future versions, our system should be able to detect such connected nodes and prefer local intra switch interaction over routed interaction.

We do not have any exact explanation why 2 compilations were faster than a single compilation. A possible explanation could be a problem with one of the node, or a better interaction of this calculation with the scheduler.

\[\text{The fastest observed speedup was 2.58 with our implementation, compared to 2.94 with dist-cc.}\]
Run with node activity  This test was performed in order to demonstrate tolerance of the system to an unanticipated user interaction. In this test, we again run a single kernel compilation on an increasing number of nodes, but the nodes start a simulated user activity during the computation. First node starts its simulated user activity after 20 seconds, then next after 40 seconds and so on, until the fifth remote node starts its simulated activity after 100 seconds from calculation start.

![Figure 5.10: A graph capturing a measured duration of a compilation on N nodes with 5 nodes performing a simulated user activity.](image)

The graph 5.10 illustrates scalability of our solution. The measurement has to start for minimum of 6 nodes, because the tests performs gradual start of a user activity on 5 remote nodes. When running with exactly 6 nodes, our measurements seems to be faster then ideal time. This is caused by delayed reaction to a local user activity, there is a 10 seconds interval in which nodes can finish remote tasks, but we did not consider those 10 seconds in a calculation of the ideal time. It is not easy to do so, because those 10 seconds are not a time when remote nodes can be fully used, but they can only finish current activities.

The main reason for this test is not to assess scalability, but to show the system is resilient to an unanticipated user activity. Some system, especially those with static scheduling policies, would not be able to quickly react on a sudden change in the environment, but thanks to the power of simple self scheduling algorithms, our system works even in case when most of the remote nodes suddenly stop contributing they computing power.
### 5.4 Measurement on heterogeneous clusters

In addition to measurements in a homogeneous cluster, we have also performed measurements in a small heterogeneous cluster of up to 4 nodes. The results of measurements were published in [A.4]. We provide a brief summary of those results here.

The measurements were performed before introduction of the measurement framework, so test definitions for those measurements are not in our repository. We also do not provide exact specification of used machines, as in this case a more important factor is a relative performance difference between the machines, and it cannot be expressed by a machine specification. Instead, relative performance is well captured by an execution time on a single machine and comparing those. When somebody is going to replicate the results, it would be probably prohibitively difficult to find exactly the same set of (random) hardware we used for testing, but any non-heterogeneous machines with similar performance ratios should give comparable results.

The measurements were again performed on Linux kernel compilation. Due to a heterogeneous nature of used computers in these tests, it is important which computer initiates calculation and the order in which they are being added (unlike as with homogeneous cluster, where this should not make a difference).

The table 5.1 summarizes performance characteristics of individual machines used during the testing. As mentioned, we do not provide exact frequency as it is meaningless for comparison of machine performance for a single application, and we rather show an execution time of compilation achieved by each of the nodes.

The figures 5.11 and 5.12 capture the basic case when we test with a single node, 2 nodes, 3 nodes etc. As mentioned, the order adding of nodes is important, so we show 2 basic cases, when we initiate the compilation on the slowest node (figure 5.12) and we are adding the nodes from the slowest to faster, and the other way when we start from the fastest node (figure 5.11) and add the nodes from the fastest to the slowest.

As the results on graph indicate both cases scale very well till 4 machines available. A percentual overhead over ideal time is much higher (around 30%) when started from the slowest machine, compared to the case when we start from the fastest machine (around 14%). This is caused by 2 factors. First, in case the calculation is started from the fastest machine, a larger amount of work is being executed locally without any clustering overheads compared to the other case. Second, the faster machine copes better with handling shared

<table>
<thead>
<tr>
<th>Name</th>
<th>Cores</th>
<th>Mem.</th>
<th>Build time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alfa</td>
<td>2</td>
<td>4</td>
<td>2:07</td>
</tr>
<tr>
<td>Beta</td>
<td>2</td>
<td>2</td>
<td>3:21</td>
</tr>
<tr>
<td>Gamma</td>
<td>2</td>
<td>2</td>
<td>3:38</td>
</tr>
<tr>
<td>Delta</td>
<td>1</td>
<td>1</td>
<td>6:32</td>
</tr>
</tbody>
</table>

Table 5.1: Parameters of the test machines. Memory is in Gigabytes, build time in format minutes:seconds.
cluster functionality, like work scheduling or file serving. The slowest machine can get overloaded and this results in higher overhead. Future optimization of our system should help to reduce this overhead.

Finally, the figure 5.13 shows the case, when all nodes start simultaneously their own calculation. This is analogous case the measurement with homogeneous cluster, when 4 nodes started the same compilation, see Section 5.3.2, with the difference that we start on all nodes and the nodes have a heterogeneous performance characteristics.

As indicated on the graph, each of the nodes computed its own compilation at the same time or a faster than it would if it did not participate in a cluster. This is expected behavior, because nodes reject any task immigration request when they are processing their own resource demanding calculations. When the fastest node is done with its own calculation, it starts accepting immigration requests from other nodes. That is the reason why all other nodes finished their calculations faster than on their own. The slowest node benefits most of the cluster, as it never finishes before the others and so it does not accept any remote tasks. Instead, because all the others finish before it, it eventually can benefit from computing power of all other nodes.

It may seem that it is not worth for the faster nodes to offer their resources to slower nodes, but it is not the case, because even the fastest node can benefit from the slowest node as shown on the figure 5.11. Generally, we can conclude that all heterogeneous nodes benefit from the cluster membership, but in a very rare case when all nodes would execute the same amount of work at the same time, the fastest machine would see no improvements.

Figure 5.11: Graph capturing a real execution time and an ideal execution time of a kernel compilation started from the node Alfa.
5.5 Predictions accuracy

In order to assess its usefulness, we have performed an isolated tests of a quality of prediction mechanism described in Subsection 3.7. We have measured an accuracy of predictions of compilation times for individual compilation tasks depending on a file being compiled. Our way to measure predictions accuracy was as follows:

1. Run once a non parallel (-j 1) kernel compilation on a local node with classification framework started in a learning mode.

2. Run repeatedly a non parallel kernel compilation on a local node with classification framework started in a learning mode, but this time compare also predictions with actual times.

We have run 5 iterations of step 2 and the results are captured in the table 5.2. As the table indicates, the predicted times differ from real times in average by roughly 6 percent after the first run and after a few more iterations we get to average difference of about 4 percent. Those are very good numbers that can give the scheduler some useful data for better planning. Of course, those numbers are not the values that would be seen when running in highly parallel distributed environment, but if they are used as a comparison metrics between individual subtasks, the absolute numbers are not necessary. The prediction estimates could be also obtained from sampling of real parallel distributed execution, but more experiments are required to evaluate quality of such a data.
Figure 5.13: Chart capturing an execution of 4 parallel kernels compilation. For each node we show a time how long would it compile a kernel on its own, and how long did it take when nodes were clustered together and each of them started its own calculation.

<table>
<thead>
<tr>
<th>Run</th>
<th>Difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st</td>
<td>5.8%</td>
</tr>
<tr>
<td>2nd</td>
<td>4.5%</td>
</tr>
<tr>
<td>3rd</td>
<td>4.6%</td>
</tr>
<tr>
<td>4th</td>
<td>3.8%</td>
</tr>
<tr>
<td>5th</td>
<td>4.2%</td>
</tr>
</tbody>
</table>

Table 5.2: An average difference of predicted values from real values in percents.
Chapter 6

Conclusions

We conclude the thesis with a summary of results, highlights of key contribution, and a suggested future work.

6.1 Summary

We have proposed and implemented a new clustering system that could be used in environments where current clustering solutions cannot be securely used. Those environments involve any computer network where computers could be administered by more than one entity. This is a very common case of industrial environments where developers often have administrator access to their machines (and often own them), but it is also becoming common in university networks, where users can bring their own portable computers and connect them to the network. Even a single computer connected to the network could compromise the whole network without proper security mechanisms.

An important advantage of our architecture over traditional dedicated architectures is that users could make clusters directly from their machines, without sacrificing autonomy or security of their machines. They can start using other machines for their own computation and in exchange offer their own idle computing resources to others. This reciprocal resource exchange could serve as a motivating factor for users to join to the cluster.

The system is since the beginning designed to be extendable to a large scale grid system. It can be either extended with a new functionality as described in Section 6.3 or it could be used as a subsystem of some other existing grid system.

We have paid a special attention to making the process of joining to cluster as easy as possible. It is sufficient to have the system installed and with default configuration a node automatically joins a cluster if it detects other nodes with the system on the same network. This could be easily extended to automatically joining over the Internet by some set of known or learned addresses, initially probably using some dictionary service. The low joining barrier could be an important factor for a system adoption.

A major contribution of our work is in proposing and demonstrating feasibility of a fully decentralized fine grained security system, that could be used in peer to peer clusters.
The implementation is specific to our system, but the idea is generic to all peer to peer computing systems and could be reused in other systems lacking this functionality like Mosix.

The proposed security system is a unique combination of local-identity-based access management with a distributed delegated access control. The local-identity-based access is conceptually easier for users and simpler to implement. The delegated access control system allows a greater flexibility for advanced users. Users can use both approaches at the same time. With our delegated access control mechanism, the users can control distribution of certificates via back propagation mechanism and trade requirements on storage space of certificates for freshness of information available on a core node.

Another important contribution for research and educational purposes is an easy to experiment scheduler implementation. Traditional clustering systems have hard coded scheduling algorithms, often residing directly in kernel. For ordinary cluster users, it is usually impossible or very hard to modify scheduling algorithms and they can influence them only indirectly by job specifications. In contrast, in our system users can easily experiment with scheduling strategies by implementing a simple class in a high level programming language. This way, users are given a power of controlling behavior of lower level layers without understanding unnecessary details of those layers.

### 6.2 Contributions of the Thesis

The key contributions of this thesis are:

1. A proposal and implementation of a secure peer to peer clustering solution, that could be used in real world environments without non realistic assumptions about environment trustworthiness.

2. A security system inspired by trust management systems, tailored for usage in peer to peer clusters, was proposed and its core was implemented for the Clondike system. A similar system could be used for any other peer to peer clustering solution.

3. A flexible scheduling framework, that is extremely easy for experiments and even non expert users can create their own customized scheduling strategies.

### 6.3 Future Work

The Clondike project is still alive, some further topics are being investigated at the moment and some other would be highly desirable provided sufficient research head count is available. We list here those topics that are being investigated now, or the topics we believe are important for future of peer to peer clusters.
6.3.1 Filesystem

The currently used filesystem is sufficient, but the caching solution used is limited. For better system performance a more sophisticated special purpose filesystem is required. The filesystem should support efficient, transparent, and coherent caching for the ONDC environment. Ideally, it should as well perform cooperative caching. The cooperative caching will be required for a larger cluster, to prevent core node overloading and as well to allow efficient usage of remote clusters connected over a low speed network with a core node. Without cooperative caching, all nodes in those clusters would have to load required binaries and libraries over the slow network, but with a cooperative caching, the loading can be performed just once and then distributed over a local network, see Figure 6.1.

Figure 6.1: First, a node B gets some file X from a node A. Afterwards, when a node C needs the file X, it could get it again from the node A, but also from the node B. Getting from the node B could be advantageous if it is closer to the node C, or if the node A is getting overloaded.

Another independent desirable feature of the ONDC filesystem is ability to reuse equal files without any transmission. Typically, in a computer laboratory or a company department, a lot of computers will share their operating system and libraries versions. In such an environment, it is not strictly required to transfer libraries between nodes, when both nodes have the same library version. A smart filesystem could detect this case, and load the library from a local filesystem instead of using remote filesystem.

6.3.2 Scheduling

Our current scheduler is designed to work well in untrusted small scale networks with no explicit knowledge about individual machines performance or task requirements. In many cases, however, some information may be either directly or indirectly available. For
example, if there is a trust between machine owners, the provided data about machine performance or task resource usage could be considered trusted. As our simple prediction framework demonstrated, some program attributes could also be automatically predicted. Our current scheduler does not take any of the task or machine properties into consideration, but in the future, it may be able to use those data to make better scheduling decisions.

A trust between nodes could be used not only for trusted information exchange, but also for a cooperative scheduling where schedulers plan tasks in a way that is most efficient for them as a group, not like our current scheduler that tries to optimize the performance always only from its own point of view.

An interesting option for further research are market-based schedulers [64]. They were already researched in the context of non-dedicated clusters [53], but existing proposed solutions required centralized components, so they cannot be directly used in an ONDC environment.

Our current system also does not address problem of free riders [51]. As shown in [64], a simple technique could be used to address the problem. The main idea is that each node keeps track about estimates of computing power granted by other nodes to it. If the node receives requests for execution of multiple remote tasks from multiple remote nodes, it gives a priority to the node that contributed most to this node. A node with a high contribution can even preempt currently running remote task with low or no contribution. Thanks to the presence of unforgeable node identities and clean design of scheduler implementation, such a mechanism could be implemented in our system in no more than a few hours.

### 6.3.3 More powerful migration techniques

There are still some technical limitations what types of programs can be migrated in our framework. The most important limitation at the moment is lacking support for migration of multithreaded applications. We already do have an experimental version of this functionality [61], but it has not yet been integrated into our main code.

The second important technical limitation is inability to migrate application with open network sockets. An often used solution is to keep traffic flowing through a core node and redirect all related traffic from/to a current execution node. This is not very efficient in terms of performance, but other solutions are problematic due to the design of currently used protocols like TCP. There were also some attempts to implement a real socket migration that would not require traffic flow through a core node [15], but this is useful and feasible only in fully dedicated clusters, where all network is controlled by a single administrative entity and all nodes and network components are expected to cooperate.

An often asked functionality in clustering systems is a support for a software-based distributed shared memory. Even though this feature is appealing in theory, in practice it turns out it does not perform well on real programs. For this reason, clustering systems typically do not support distributed shared memory, and we currently do not plan adding this functionality.
6.3. **FUTURE WORK**

### 6.3.4 Introduce reputation into security

A possible improvement to our security system is incorporation of reputation into a security system as done in some other trust management systems [84]. This would allow users to distinguish between unknown nodes and grant nodes with higher reputation higher privileges than to other unknown nodes with no or bad reputation history.

The reputation could be used to increase a size of pool of available nodes for computations that require at least minimal degree of trust. It can also be used to evaluate trustworthiness of information coming from remote nodes, and if the information is considered trustworthy, it could be passed to scheduler for further processing.

### 6.3.5 Interconnection of large networks

Currently, every core node tries to connect itself to a preconfigured count of other nodes and if a number of passive remote nodes falls under a certain threshold, it tries to connect more nodes. If all nodes are acting as core and detached nodes at the same time, this ultimately leads to a topology, where all nodes are connected together. This is not a problem in small scale local networks with up to a few tens of nodes, but it is not a usable strategy for large scale networks spanning over the Internet.

When the system is used in a more complex network structures, possibly spanning over the Internet, it must address the following topics:

1. **Node discovery:** The nodes in the system should be able to find nodes residing on other networks, not only locally as it does now.

2. **Awareness of network topology:** Current system ignores locations of nodes in the network as we have been using it only on local networks. However, if remote networks are used, the system must be aware of those differences and prefer interaction with co-located nodes.

3. **Addressing routing and reachability restrictions:** In practice, most of the networks reside behind a firewall, and often they are in addition hidden by a network address translation \texttt{NAT} gateways. The system should be able to work in those environments.

The proposed solution could be based on peer to peer networks algorithms, as they traditionally address those problems. An interesting solution is the java based JXTA framework, that deals exactly with this problem.

### 6.3.6 MPI integration

MPI is a very popular system for writing distributed applications. Running MPI application on our system would not be efficient. The reason for low efficiency is the lack of support for remote sockets. All communication between nodes have to go through a core node and this would have very negative impact on performance.
A possible approach how to address this problem would be implementation of support for remote sockets. However, since MPI is so popular, it seems appropriate to implement a special support for MPI systems directly in the system. The integration should consist of 2 parts. First, an ONDC specific transport method between MPI nodes should be implemented. OpenMPI implementation seems to be a good candidate for this extension as it supports custom transports. The main motivation for a specific transport layer is to allow easy migration of end points, as this would be problematic with standard TCP/IP based transport method. Second part of integration should reside in specific scheduling. The system could easily detect through `exec` call hook that a new MPI application is being started. This could trigger special scheduling rules, that would optimize placement based on MPI specific requirements.

\footnote{For example, MPI tasks typically expect all nodes to handle roughly the same amount of work, which is not a generic case for all tasks being run on ONDC.}
Bibliography


Publications of the Author


The paper has been cited in:


Appendix A

Cluster APIs exposed by kernel

There are 2 main APIs exposed by kernel code to the user mode - pseudo filesystem based and netlink based APIs. Here we briefly describe the exposed functions.

Pseudo file-system based API

Pseudo filesystem based API was the first kernel API used in Clondike, created by the original authors of the system. It was the simplest way how to expose basic information about a cluster and give some control to users over the core cluster features. The API is suitable for reading operations, but less suitable for write operations as there is no standard way how to pass structured data. To read from the API any standard Linux command line for reading files could be used, for example the `cat` command; for writing the `echo` command with redirection to a specified pseudo file should be used.

The pseudo filesystem based APIs have to be mounted at some location in filesystem to become accessible to users. Clondike uses its own custom mount point at directory `/clondike/`. When the filesystem is mounted, all files under this folder are pseudo files or pseudo directories. There are 2 main subdirectories under the root folder - `ccn` for API related to the core node functionality and `ppm` related to the detached node functionality. Both of these APIs can coexist on a single node, but node owner could decide to run just one of them (i.e. enable for example only core node functionality). We describe both of these APIs separately.

When using a user mode director, the write files should not be used directly, as they are instead used by the director component.

Core node API

Initiate listening `/clondike/ccn/listen` pseudo file is used to initiate listening on a specific protocol and port combination. The provided argument is in format `<PROTOCOL> :<IP-ADDRESS> :<PORT>`. If a computer has multiple network interfaces, it is possible to initiate multiple independent listenings, each on a different IP address.
APPENDIX A. CLUSTER APIS EXPOSED BY KERNEL

Stop listening  /clondike/ccn/stop-listen-one or /clondike/ccn/stop-listen-all pseudo files could be used to stop one of the listenings, or all of them. The first command takes as an argument an index of the listening to cancel, with 0 being the first initiated listening. This command could be used when user wants to stop accepting new nodes, but still keep existing connections running.

List listenings  /clondike/ccn/listening-on/ directory contains a list of current listening. Each of the listenings creates its own directory in listening-on directory, named listening-<INDEX-OF-LISTENING>. Each of these directories will have subdirectories sockname (containing an argument passed to listen) and arch (containing the protocol name used for connection).

Stopping the core node functionality The best way to stop the core node functionality is to unload cluster modules, this will take care of all required steps like unmounting mounter filesystem, migrating home all cluster processes etc. In case the unloading is not an option, the /clondike/ccn/stop method could be used.

Listing all connected detached nodes  /clondike/ccn/nodes directory contains all connected detached nodes. For each detached node, there are 2 directories, one named by the detached node IP address and the other named by the index of detached node, the first detached node connected has index 0. Those directories represent migration managers (see Section 4.4) associated with the detached nodes. In each directory, there is one file per process that is currently emigrated to a corresponding node, the files have name equal to the local process identifier.

Each of the node directories contains also the following files:

state File containing a current connection status of the remote node. Only nodes in status Connected are fully operational. Other states represent some transient mode, either when the connection is being initiated or terminated.

stop Writing anything to this file will initiate node disconnection. First, all remote processes are requested to migrate back. After some timeout, pending processes are killed and finally the connection is terminated. This call is synchronous.

kill Similar as the stop command, but it does not try to migrate processes home, but instead immediately tries to kill them all.

load Obsolete, was intended to report detached nodes load, but did not ever work.

connections/ctrlconn Contains 3 files, arch specifying the protocol used for connection, sockname containing an IP address and port combination where the core node is listening, and peername containing an IP address and port combination of detached node’s connection endpoint.
Requesting preemptive process migration  The preemptive process migration can be requested by writing to the /clondike/ccn/mig/emigrate-ppm-p. The format of string is <PROCESS-ID> <NODE-ID>, where PROCESS-ID is a pid of process to be migrated and NODE-ID is an index in node list of a remote detached node, where the process should be migrated. The request is asynchronous, i.e. the caller returns immediately and the migration (may) happen at some later time. Return from the call does not guarantee a success of the migration.

Currently, only processes that are running on the home node can be migrated this way; if a process is at the moment running on a detached node, it has to be migrated back before it could be emigrated to some other detached node. Note, that non preemptive migration cannot be requested by the pseudo filesystem API.

Requesting migration back to home node /clondike/ccn/mig/migrate-home pseudo file is used to migrate back a process that currently runs on some detached node. An argument to this file is a process id (as seen by core node) of the process to be migrated home. Similarly as emigration request, this request is also asynchronous.

Listing emigrated processes /clondike/ccn/mig/migproc directory contains one directory per each process originating from this core node that is currently running on a remote node. The directory names equal to a local process id of those processes. In each of these directories there is a file remote-pid containing the process id of a corresponding guest process running on a remote node and a symbolic link migman leading to a directory representing a migration manager associated with the process.

Detached node API

Connect to a core node  /clondike/pen/connect file is used to request a connection to a core node. The format of argument to be written is <PROTOCOL>:<IP-ADDRESS>:<PORT>. The call is synchronous so a successful return from writing means the basic connection was established. It does not guarantee that all other supporting connections like filesystems will succeed, so the connection still may get terminated shortly after being initiated.

Disconnecting from core nodes The best way to terminate all detached node functionality is to unload kernel modules. This will attempt to migrate back all processes and then terminate all connections. Alternatively, file /clondike/pen/stop can be used.

Listing all connected core nodes /clondike/pen/nodes has nearly the same structure as a corresponding folder for core nodes. Each of the nodes here represents a remote core node, that is connected by a detached node represented by current machine. stop and kill files have a similar meaning, they do request migration from the current node back to core node and then terminate the connection. A special migrate-home-all is used to
asynchronously request migration of all remote processes from current detached node to their home core nodes. It does not terminate the connection.

**Requesting migration back to home node**  
/clondike/pen/mig/migrate-home pseudo file is used to asynchronously request migration of a process back to its home core node.

**Listing immigrated processes**  
/clondike/pen/mig/migproc directory has a similar structure as migproc for core nodes, but it contains processes that are at the moment running on current detached node.

**Netlink API**

The netlink [37] based API was introduced after the pseudo file-system API and can eventually take over all its functions. However, the pseudo filesystem read API should stay as it provides a simple way how to find out basic information about the cluster state just from command line, without relying on any user mode component.

There are 2 basic sets of messages - notifications and commands. Notifications are send by the kernel mode code to the user mode, while commands are send by the user mode to the kernel code.

**Commands**

**Register for listening**  
This command is invoked automatically by the user mode C api when it is initiated. It is required as the kernel Netlink API does not know where to send all notification messages. The registration does register current process identifier as a receiver of kernel initiated Clondike related netlink messages.

**Send a generic message**  
Used to send any message to a connected core or detached node. The message content is opaque to the kernel mode. It is used for example for heart beat based checks.

```c
int send_user_message(int target_slot_type, int target_slot_index,
                     int data_length, char* data);
```

- **target_slot_type** Either 0 to send a message to a core node or 1 to send a message to a detached node.
- **target_slot_index** Index of a core node or a detached node in nodes directory.
- **data_length** Length of a data being sent.
- **data** Any data to be send of at least data_length length.
Notifications

For notifications, instead of their invocation signature, we describe rather their invocation callback signature. The callback is invoked in user mode and it has to be first registered by a corresponding registration method.

Non preemptive migration This callback is invoked always when an exec command is invoked on some local process. The callback can decide, if a non preemptive migration should be performed at this point and to which node should the process be migrated. Even if the callback decides the process should migrate, it does not guarantee a success, there could be some transient network failures, or a destination node could simply refuse to accept the process. In this case, the process is normally started on the core node.

typedef void (*npm_check_callback_t)(pid_t pid, uid_t uid,
   int is_guest, const char* name, struct rusage *rusage,
   int* decision, int* decision_value);

typedef void (*npm_check_full_callback_t)(pid_t pid, uid_t uid,
   int is_guest, const char* name, char** args, char** envp,
   int* decision, int* decision_value);

There are 2 version of callback. First callback is called without arguments and the environment variables. In case the decision could be made without these argument, the second callback is never called. However, the first callback could decide, that it needs access to arguments or environment variables and in this case it returns a special response requesting a second full callback. The reason for this separation was to reduce a volume of data that needs to be serialized and transferred on each of the exec calls, assuming that often just a process name is sufficient to decide about the best migration strategy.

pid Process identifier of a process performing the exec system call.

uid User identifier of owner of the process.

is_guest True, if the process is currently running as a guest process on a detached node.

name Name of the binary being executed.

rusage Resource usage structure associated with the process.

args Arguments of the process.

envp Environment variable os the process.

decision Output variable, filled by the callback. Possible values are 0 (do not migrate), 1 (migrate to a remote node), or 2 (migrate back home). If args or envp is required, return value should be 5 to request a second callback with those arguments filled.
**decision_value** Output variable, required only if decision was set to 1. In that case, this argument specifies a node index where the process should be emigrated.

**Immigration request** This invoked on detached node, when some core node tries to migrate a process to that node. The detached node has an option to either accept the request or reject it. The request is sent before any other migration steps are initiated.

typedef void (*immigration_request_callback_t)(uid_t uid, int slot_index, const char* exec_name, int* accept);

- **uid** User identifier of owner of the process.
- **slot_index** Index of the migration manager that is requesting the migration.
- **exec_name** Name of the binary being migrated.
- **accept** Output variable, set to 0 if the callback wants to reject the request, some non-zero number otherwise.

**Immigration confirmation** Invoked on a detached node, when a process immigration was successfully completed. After this callback, the immigrated guest process is fully running.

typedef void (*immigration_confirmed_callback_t)(uid_t uid, int slot_index, const char* exec_name, pid_t local_pid, pid_t remote_pid);

- **uid** User identifier of owner of the process.
- **slot_index** Index of the migration manager that handled the immigration.
- **exec_name** Name of the binary being migrated.
- **local_pid** Process identifier of the guest process on the detached node.
- **remote_pid** Process identifier of the shadow process on the core node.

**Emigration failed** Called on a core node, when a process emigration has failed. After this callback an execution of process continues on a core node.

typedef void (*emigration_failed_callback_t)(pid_t pid);

- **pid** Process identifier of a process that failed to emigrate.
**Migrated home**  A notification callback invoked when a process has been successfully migrated home. It is invoked both on core and detached nodes.

```c
typedef void (*migrated_home_callback_t)(pid_t pid);
```

`pid`  Process identifier of a process that failed to emigrate. On both core and detached node this is a process identifier local to this node.

**Node connected**  Invoked on a core node, when a detached node tries to connect. The callback is invoked when the main kernel to kernel channel is opened. Before anything else can happen, the remote node needs to be admitted by this method. The detached node could provide some authentication data and the callback decides whether the detached node should be admitted or not.

```c
typedef void (*node_connected_callback_t)(char* address, int slot_index, int auth_data_size, const char* auth_data, int* accept);
```

`address`  IP address of the detached node that is trying to connect.

`slot_index`  An index of a newly created migration manager associated with the node being connected.

`auth_data_size`  Length of the authentication data.

`auth_data`  Authentication data, opaque to the kernel mode code, it is fully up to the user mode code to decide what kind of data to send and verify.

`accept`  Output variable, set to 0 if connection is rejected or to 1 if it is accepted.

**Node disconnected**  Called, when a peer node disconnects. It is called both on core and detached nodes.

```c
typedef void (*node_disconnected_callback_t)(int slot_index, int slot_type, int reason);
```

`slot_index`  An index of a migration manager associated with the disconnected node. Could be both manager of a detached node or of a core node, this is specified by the next argument.

`slot_type`  1 if a remote detached node was disconnected, 0 if a remote core node was disconnected.

`reason`  Is set to 0 if the node disconnect was initiated locally (for example node was declared as dead, or it was not useless any more), and set to 1 if a remote node requested the disconnection.
Task exit  Invoked when a process execution ends. Invoked on both core and detached nodes for all processes.

typedef void (*task_exitted_callback_t)(pid_t pid, int exit_code, struct rusage *rusage);

pid  Local process identifier.
exit_code  Exit code of the process.
rusage  Kernel resource usage structure associated with the process.

Task fork  Invoked when a process fork is executed. The callback is executed at the end of the fork system call, after the child process has been successfully created. Invoked on both core and detached nodes for all processes.

typedef void (*task_forked_callback_t)(pid_t pid, pid_t ppid);

pid  Local process identifier of a newly created child.
ppid  Local process identifier of a parent process being forked.

Check permission  A callback from a kernel when an access permission check has to be performed. The kernel does not contain any high level security logic, so it needs to query user mode for evaluation. The callback propagates all flags defining a permission as described in Section 3.4.2.

typedef void (*check_permission_callback_t)(pid_t pid, int permission_type_size, char* permission_type, int operation_size, char* operation, int object_size, char* object, int* allow);

pid  Local process identifier of a process performing a privileged action.
permission_type_size  Length of a string representing a permission type being requested.
permission_type  Permission being requested, for example fs
operation_size  Length of a string representing an operation to be performed.
operation  Operation for which the permission is being evaluated, for example read.
object_size  Length of operated object string representation.
object  Object being access, for example /home/user/some.file.
allow  Output variable, a decision of the security system whether to allow the call (1) or reject it (0).
Receive a generic message  This callback is counter part of send generic message command; it is invoked when a generic message is delivered to either a core or detached node.

typedef void (*generic_user_message_callback_t)(int node_id, int slot_type,
        int slot_index, int user_data_size, char* user_data);

node_id  Obsolete, not used any more.

slot_type  Either 0, when a message arrived for a core node migration manager, or 1 when it arrived for a detached node manager.

slot_index  Index of a receiving core node or a detached node migration manager in the nodes directory.

user_data_size  Length of a data being received

user_data  The data being received.