Czech Technical University in Prague
Faculty of Information Technology
Department of Computer Systems

Towards a decentralised peer-to-peer cluster

by

Josef Gattermayer

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

Dissertation degree study programme: Informatics

Prague, September 2017

Supervisor:
prof. Pavel Tvrdk, Ph.D.
Department of Computer Systems
Faculty of Information Technology
Czech Technical University in Prague
Thákurova 9
160 00 Prague 6
Czech Republic

Abstract and contributions

The Clondike - CLuster Of Non-Dedicated Interoperating KErnels - is a long term research project of a non-dedicated cluster architecture at the Faculty of Information Technology of CTU in Prague. The thesis deals with its conversion into a Peer-to-peer (P2P) scalable cluster where single nodes can share their computing resources between each other. P2P design offers many benefits over a single-master or a multi-master architecture in terms of scalability, reliability and independence. On top of that every participating node can use computing resources of other nodes as well.

None of the analyzed existing clusters provides the same functions as we implement to Clondike. Clondike operates on the kernel level thus unmodified system processes can be migrated. Efficiency of this universal approach is comparative to task-specific application frameworks what we verify on a set of measurements on different types of clusters.

An important component of every distributed system is a protocol that connects new nodes and keeps communication between the existing ones. We present an algorithm inspired by protocols used in P2P file-sharing networks. This algorithm reaches a logarithmic scalability and allows Clondike to scale as a global cluster.

Clondike is an open source project so we assume that there will be modified Clondike clients that will try to exploit resources of other nodes. We introduce a service blockchain network for off-chain Clondike computations. This network is used to log and verify all transactions so a reputation of each node can be evaluated and unfair nodes can be eliminated.

This thesis presents a working version of the proposed Clondike cluster.

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

1. Evaluation of existing peer-to-peer Clondike cluster in laboratory and real-world environments.
2. Introduction of a scalable peer-to-peer inter-node communication and bootstraping protocol.
3. Introduction of a blockchain service network for off-chain computations.

Keywords:
Cluster computing; non-dedicated cluster; distributed-networks; peer-to-peer; scalability; decentralized systems

Acknowledgements

First of all, I would like to express my gratitude to my dissertation thesis supervisor, prof. Pavel Tvrdik. He has been a constant source of encouragement and insight during my research and helped me with numerous problems and professional advancements.

Special thanks go to the staff of the Faculty of Information Technology, 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.

Finally, my greatest thanks go to my family members and princess Kaja, for their infinite patience and care.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

There are many workstations with unused computing resources. The original aim of Clondike project was to utilize unused computing resources in university and corporate local networks - but in a purely decentralized manner.

Contemporary CPUs partially eliminate the main problem of idle workstations - a power consumption - with energy-saving mechanisms and in this perspective it may seem there is no demand for a such cluster. But with a rise of peer-to-peer (P2P) systems like BitTorrent [5] the interconnection of computers became global and offered a new market with new possibilities. The decentralized architecture of Clondike became a huge advantage in designing a solution for this market.

The current Clondike aim is to make a global P2P cluster that would share computing resources between its users. The main difference from existing grid systems is that participating users can benefit from a membership in the Clondike cluster as well and are free to use computing power of others.

Clondike is also specific as it operates on the kernel level - this architecture allows to migrate unmodified system processes. With this approach a universal supercomputer for each user of the cluster is formed. This is unique as other networks with the same P2P architecture can not migrate unmodified system processes.

Current market data show an enormous demand for P2P solutions across all sectors. For example three leading projects for P2P storage (that are still in development and not production-ready) have a sum of their capitalization $326,546,064 [6].

1.2 Problem Statement

There are 3 main issues to be solved. Limitations and possibilities of current Clondike version must be researched as the first ground step. The next step is to find a real use

case that would verify the usability of our work. The later work will focus on research of specific issues associated with the global peer-to-peer cluster goal.

1.2.1 Clondike performance scalability on a real use case application

At first, overall performance and scalability of existing Clondike cluster must be evaluated. Clondike was created as an academic project and it is crucial to find out if a such general-purpose cluster can compete against task-specific frameworks. Results on laboratory tasks as Mandelbrot set computations are on thing, but only a real-use case proves viability of a project. Results must be evaluated on different cluster types in order to simulate a real operation. Finding a real use case for Clondike is a crucial step and only that will make possible a further research.

1.2.2 Scalable decentralized node discovery and bootstrapping system

The second issue is an node-discovery, bootstrapping and inter-node communication. On a small scale (laboratory environment) trivial algorithms can be used. But to fullfill the goal of a global large-scale cluster a scalable algorithm must be researched and implemented.

1.2.3 Unfakeable decentralized system for node reputation evaluation

The last issue is a reputation scoring of single nodes of the cluster. There is no central authority to identify fair and unfair nodes. But since source code of each node can be modified we can expect abusive or unfair nodes will show up. Each node must have a defense mechanism to identify these nodes.

1.3 Goals of the Dissertation Thesis

1. Evaluate scalability of existing Clondike solution on a real use case in different environments.
2. Design, implement and evaluate a scalable node-discovery, bootstrapping and inter-node communication protocol.
3. Design, implement and evaluate a decentralized scoring system of single nodes in order to guarantee fairness among the cluster.

1.4 Main contribution of the thesis

This thesis is not only theoretical but all researched algorithms and principles are also implemented, tested and evaluated. Clondike is a complex software project and its imple-

mentation consists of a patch on the kernel level, an application interface and a high level scripting framework. Source codes of implementation work are freely available at [7].

Research work consists of contributions to three main topics:

1.4.1 Clondike performance scalability on a real use case application

We present discussions on utilization of existing Clondike Vanilla. Clondike is a universal cluster, where a subject of migration is a system process. It can run any kind of job, but not all jobs run effectively. This is mainly because of a communication overhead and a relative slow interconnect network (compared to a local disk access). Synthetic jobs which utilize mainly CPU and last for minutes show always great results in therms of scalability and efficiency. The goal of Clondike project is to develop a cluster that can be used in production, not only as a research project. We present a real-world use case that utilizes capabilities of Clondike cluster.

The use case presented in this thesis is a distributed code compilation. From the wide-range of projects we run our tests and benchmarks on a compilation of a Linux kernel. It consists of a compilation of thousands of source files - from the smallest ones (that do not run effectively due to migration overhead) to the big ones (that run effectively). This ensures us that the results are not specific for this job but should be relatively the same for other software projects as well.

We introduce measurements of a distributed kernel compilation on two types of cluster. A homogeneous one, where are all nodes the same (in terms of system resources). And a heterogeneous one, where single nodes differ and correspond more to a real-world use case. Clondike is compared to a task-specific project distcc [8]. Distcc is a distributed source code compiler that runs on an application level. The very same distributed kernel compilation is run on the very same nodes once with Clondike and once with distcc. We compare scalability of both approaches. On the homogeneous cluster Clondike scales almost identically to distcc which is a very positive result for a general-purpose cluster. The heterogeneous cluster shows limitations of current round-robin-based Clondike scheduler and brings our attention to a need of a completely new approach in designing a new service layer of Clondike.

1.4.2 Scalable decentralized node discovery and bootstrapping system

Previous work proved that there exists at least one real use case where Clondike scales as well as specific application frameworks. In order to make possible a scalability along multiple networks we present a new protocol for a node discovery and a node bootstrapping. Clondike is a non-dedicated cluster so single nodes can join and leave a cluster at any time. This is a specific requirement as other researched networks have mandatory routines that must be run before a node can leave a network.

The only protocol that scales logarithmic and does not require any routines on a node disconnection is protocol Kademlia. Kademlia works on a principle of distributed hash

table. Each node keeps a connection to a specific number of nodes and each node has a unique ID in a fixed space of identifiers. This ID identifies a position of a node in the space of identifiers. Even if a node does not have a direct connection to a node it knows which of its directly connected nodes is the most nearby to the target node.

This protocol was originally developed for the BitTorrent network [9]. BitTorrent is a peer-to-peer network where single nodes share data among each other. It succeeded as a global network with millions of users that is completely independent on any 3rd party - our goal with Clondike is the same.

Kademlia protocol has many implementations, but its use case is different from Clondike. Clondike is a computing cluster, not a data-exchange network. We present requirements of a protocol that is working on the same principles but specific to a P2P cluster use case. This protocol is implemented in Ruby language and integrated into Clondike cluster.

Results of the implementation are verified on two different measurements. The first measurement is run on real hardware nodes that are running an unmodified Clondike cluster. This measurement verifies correctness of our implementation. The second measurement is run on a large scale. We allocate 4096 virtual machines at the IBM Bluemix [10] platform. Clondike is implemented on a kernel level so it can not run on a platform that does not allow modifications of the kernel. Most of infrastructure-as-a-service platforms (including IBM Bluemix) do not allow kernel modifications. In order to be able to run a large scale measurement we present a simulated Clondike kernel that runs on an application level. This simulated kernel can be used by any platform. Results of measurement on the 4096 node cluster confirm a logarithmic scalability of the new bootstrapping protocol. The 4096th node needs to exchange only 24 communication messages with other nodes of the cluster in order to join the cluster.

1.4.3 Unfakeable decentralized system for node reputation evaluation

Bootstrapping and node discovery can be solved in a similar way among multiple distributed networks. But this is not a case of a reputation scoring. BitTorrent network was an inspiration for node-discovery principles used in Clondike. But if we tried the same approach with reputation scoring we would not succeed. In BitTorrent network single nodes exchanges data between each other. If a node A has a chunk of data of a file that node B does not have, it can exchange if for another chunk of data that has node B and node A has not. The motivation of both nodes is the same - get a complete file. This principle can not be used in a computation cluster as each node needs to solve its own problem.

Clondike is an open source project so it is easy to start a node with its own version of Clondike. This modified node can be programmed in a such way that is uses computing resources of other nodes but does not contribute any computing resources to a cluster. There are many other use cases where a node can exploit a cluster. We define them and call nodes that fullfill such behaviour Vermin nodes. The goal of each node of a Clondike cluster is to identify a Vermin node and do not provide it with any computation resources. Since there is no central authority each node is responsible for itself.

This is called an open multi-agent system. There exist multiple solutions of this problem and we provide an overview of the most referenced ones. The core principle of these existing algorithms of this problem is that each node builds a network of nodes with some level of trust. This would definitely work with Clondike as well but we present a simpler architecture that would as well help to solve other unresolved issues such as load balancing.

Our source of inspiration for this architecture is from a completely different area. Cryptocurrencies such as Bitcoin [1] have a solution for the very same problem. Single participants of the ecosystem exchange assets and there is no central authority that would confirm their transfers. Each participant can claim a fake transfer and it is up to other participants to identify that. We look at computing resources as an asset as well yet we do not trade it.

The core component that solves this problem in the cryptocurrency world is a technology called blockchain [3]. Single assets are packed in transactions and each transaction is verified by a majority of nodes (more specifically by a majority of the computing power of these nodes). We present this principle more in detail. The main result is that what has been once written into blockchain can not be deleted or altered.

Our idea is to use a similar underlying technology and record there all transactions. By a transaction we mean all states of a process migration that is happening inside a Clondike cluster. We present a service blockchain network that works as a distributed log system. All transactions are verified by a majority of nodes and remain immutable in the log system. Each node can evaluate a behaviour of each node of the cluster from the distributed log system.

We present a single metrics that is represented by (only positive) Kudos value. This metrics counts Kudos of single nodes - the more positive actions a node does for other nodes of a cluster the more Kudos it has. Kudos value of both nodes is compared at the beginning of a process migration. If a respected node (with higher Kudos value) receives an immigration request from a node with low respect (zero or lower Kudos value) it may decide not to peer with it.

All computations remain off-chain and the blockchain-based distributed log system acts as a service network for a Clondike cluster. This principle radically simplifies the open multi-agent system problem and offers a whole new framework to solve another issues such as load balancing.

1.5 Structure of the Dissertation Thesis

The dissertation thesis is organized into 7 chapters as follows:

1. *Introduction*: Describes the motivation behind our efforts together with our goals.

2. *Background and State-of-the-Art*: Introduces the reader to the necessary theoretical background and surveys the current state-of-the-art. Related work is considered from different points of view - system clusters, specific systems and blockchain networks.

Blockchain networks section presents an exponentially growing sector of off-chain computing systems that are turned into an on-chain economy.

3. *Clondike performance scalability in homogeneous environment*: Presents results of measurements in a lab environment. Scalability limits of Clondike Vanilla are evaluated on a real use case scenario.

4. *Clondike performance scalability in heterogeneous environment*: Presents results of measurements in a mixed environment that is close to a small business network.

5. *Scalable decentralized node discovery and bootstrapping system*: Introduces the reader to bootstrapping in decentralized environments and presents our protocol. Bootstrapping protocols of contemporary distributed networks are analyzed and the best protocol is chosen and modified for implementation to Clondike Chameleon.

6. *Unfakeable decentralized system for node reputation evaluation*: Introduces the reader to trust and reputation in decentralized environments and presents our new architecture. Existing algorithms of the open multi-agent system are presented and discussed. All these algorithms were researched prior a new technology on the distributed scene arrived - the blockchain. Our architecture benefits from blockchain technology and presents a whole new approach that dramatically reduces the complexity of the multi-agent system.

7. *Conclusions*: Summarizes the results of our research and the status of project Clondike. Possible ways of further research are presented as future work.

Chapter 2

Background and State-of-the-Art

2.1 Theoretical Background

Clustering plays an important role in today's computer systems. All most powerful computers are in fact just clusters of semi-commodity hardware connected using a network.

The problem with these high-end cluster systems is that they require fully dedicated nodes. We are trying to overcome this limitation, so we could be able to achieve a higher efficiency of the whole network infrastructure by simply using waste computing power of ordinary workstations.

Although there are some projects with similar goals, our project Clondike goes further. Our goal is to create a universal non-dedicated peer-to-peer cluster, so that every participating node can benefit from a membership in the cluster.

2.1.1 Clondike

Clondike stands for CLuster Of Non-Dedicated Interoperating KErnels and is being developed at the Faculty of Information Technology, Czech Technical University in Prague.

The initial goal [11] of the Clondike project was to design a new type of a cluster of Linux machines. It should be capable of utilizing standard Linux machines as its computational units, while still maintaining the illusion of a powerful Single System Image (SSI). The unique feature of the Clondike system is its ability to integrate the workstations even if they are not fully dedicated to the system. They could still be used and administered by their users/admins and offer their computing performance to the cluster only when they become less utilized [12]. Clondike operates on the kernel level, so it is fully transparent to a user - it does not require any additional tools on a user level to handle the process migration.

Currently, the Clondike system is being extended into a full P2P operating-system-level cluster [13]. In Vanilla version, each participating user can form his own virtual cluster and use the computing performance of the other workstations for his own computing. The P2P architecture makes the system fault-tolerant, since there is no single point of failure,

Release name	Release goal
Vanilla	Peer-to-peer cluster
Chameleon	Global scalability
Coypu	Reputation scoring

Table 2.1: Clondike releases

all machines are fully autonomous. In addition, the new architecture makes Clondike more attractive to end users than non-P2P clustering architectures. Clondike users not only contribute computing performance of their machines, but they can also use computing performance of the other machines for their own computations.

Since every computer in a Clondike system forms its own administrative domain, the Clondike system goes beyond boundaries of traditional clustering systems. It is similar to the architecture of a P2P-based grid system.

Source files can be downloaded from [14].

Table 2.1 defines for the purpose of this thesis release names that are related to single project steps. Subject of this work are releases Chameleon and Coypu.

Results of work discussed in this chapter were presented at International Conference on Parallel Processing (ICPP) conference workshop in Bristol and will be published in conference proceedings.

2.1.2 Clusters vs Clouds

Clondike doesn't fit in any existing category of distributed computing systems. By making a Unix system process a migrate-able entity, it is closer to grid systems. But these systems have one central management point, whereas Clondike is designed as a P2P system. This P2P approach is used in some cloud systems, as shown in the following section.

In general, many big grid projects have disappeared in last years and we have been watching a rise of cloud computing. Grid systems remain mainly for scientific computations, whereas cloud systems act as internet infrastructure. Clondike aims to be in a category for its own, to extend an existing workstation with computing power of others.

2.1.3 Blockchain

Blockchain is a key component of contemporary cryptocurrencies such as Bitcoin [1], Litecoin, Ethereum. Blockchain solves the main problem of digital assets the double-spend problem. Each digital asset such as file, email or an array can be copied and a spectator can not determinate the origin and the copy without a 3rd party. In P2P networks there is no such authority. The ownership of an asset is recorded in a public ledger. This ledger is confirmed by the whole community of the system, so the trust is not needed between two parties, but between one party and the whole system.

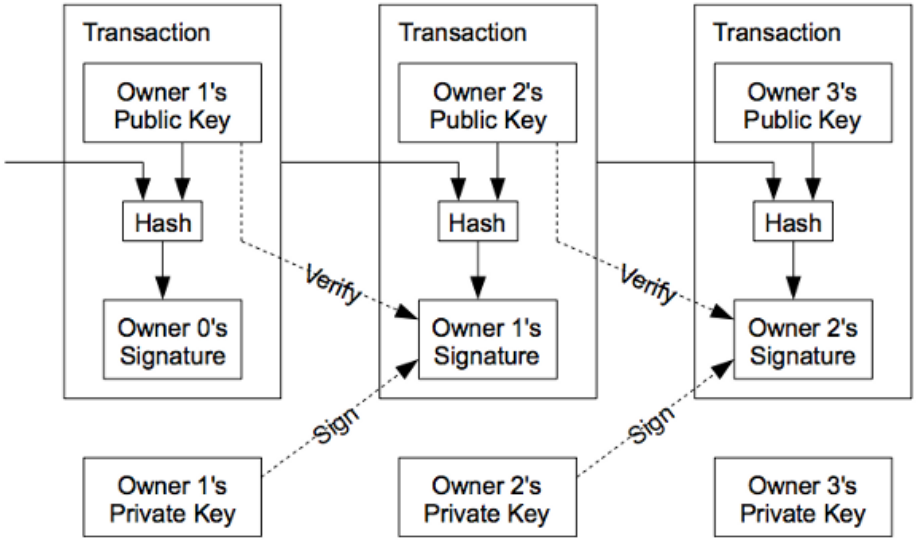

Figure 2.1: Blockchain [1]

The basic principle is based on asymmetric cryptographic functions. Each transaction is signed by its owner. In order to be a valid transaction of the public ledger, the transaction must be hashed and the hash must be included in the next transaction, see Figure 2.1.3.

In real blockchain implementations (such as Bitcoin) transactions are organised into blocks.

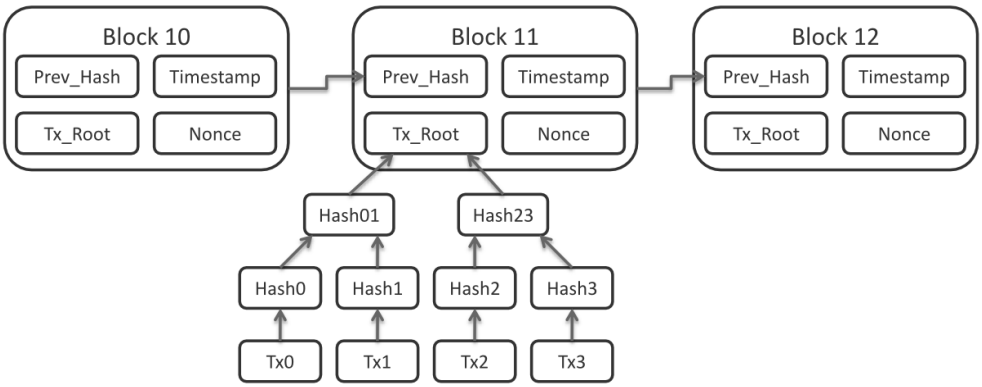

Figure 2.2: Bitcoin block [2]

Each block contains transactions since the previous block (Figure 2.2). Each block of the blockchain confirms identity of the previous block all the way back to the very first, the `genesis block`. Single blocks are identified by their hash and the hash is also a string that verifies their integrity. Finding a new valid block is a non-trivial task. The hash of the resulting block must contain a specified number of leading zeros. The only way to achieve this is by trying and modifying data in a block nonce value (sometimes called

salt). This operation is very compute-intense. The motivation for single nodes of the network to calculate hashes of previous blocks and to find new blocks is based on a reward mechanism. Each transaction can contain a reward, that goes to the block owner that includes this transaction in its block.

There is a scenario that can happen when there are discovered two new blocks that both point to the same parent and both have a valid hash with a specified number of leading zeros. Both new blocks can contain the same transaction and cause a double-spend problem. To eliminate this problem, only one block can be included into the chain as valid. The rule is that the valid branch is the longest one. This forces single nodes that compute hashes to decide between one of the two blocks to calculate a new hash from. The block that has more computing power behind it wins this race and is first verified by a child block. This child block becomes the longest branch of the chain and other nodes will be motivated to compute a new hash for a new block out of it. More illustrates Figure 2.3.

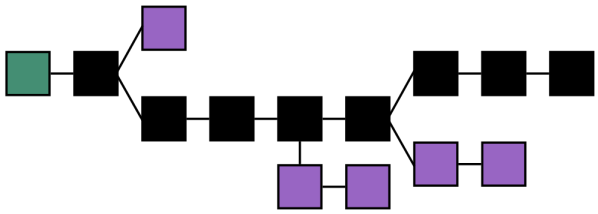

Figure 2.3: Multiple blockchain branches [3]

Who controls the majority of computing power controls the blockchain. There are known issues where an extensive rollback of many transactions was done. One example for all hardfork [15] that happened 18. October 2016. Ethereum is a young cryptocurrency marked as feature-coin. It can be extended by anyone by writing and submitting a contract to Ethereum blockchain. There was a bug in one of the most popular contracts DAO [16] [17] that led to an exploit. Total value of $50M was stolen by anonymous hackers and the Ethereum community decided to roll-back this transaction as it was found out. All child blocks from block 1920000 were thrown away and the hash power of the community build an alternative branch from previous block before the attack happened and this branch became the major branch.

2.2 Previous Results and Related Work

2.2.1 Previous Results

Vanilla Clondike 2.1 was released before this thesis. It was a result of work of two Ph.D candidates Martin Kacer and Martin Stava, both supervisored by prof. Pavel Tvrdik. Fundamentals for unmodified process migration were implemented and a peer-to-peer design was introduced.

2.2.2 Kerrighed

The Kerrighed [18] project has been started by Christine Morin in 1999 at INRIA, the French national lab for research in computer science. Since 2006, Kerrighed is a community project developed by Kerlabs, INRIA, partners from the XtreemOS consortium.

2.2.2.1 Architecture

Kerrighed has an architecture similar to Clondike. It is a cluster of standard PCs and acts as a Single System Image (SSI). Same as Clondike, Kerrighed is realized as a patch to Linux kernel. Unlike Clondike, it creates one SSI among all cluster nodes, Clondike creates a SSI for each participating node.

Clondike is fully transparent to a user (i.e., a user does not have to run commands in any other way he used to and they are automatically run on the cluster), Kerrighed creates a virtual Linux container (lxc) that the user must login to to use the SSI cluster. This virtual Linux container is accessible on any cluster node using ssh, it runs on port 2222. Processes that run outside the container (i.e., on a cluster node in its OS) are not run on the cluster. See figure 2.4 for details.

From version 3.0.0., Kerrighed can only be booted using `PXE` and the root file system is on `NFS`. Using this setup, all nodes have the same filesystem. Even some commands may be run locally, in fact, they are fully dedicated to the cluster. Clondike can boot from `PXE` as well, but every node has its own filesystem connected over `NFS`. It can boot from a local hard drive as well. This option is no longer supported in Kerrighed.

2.2.2.2 Current state

Kerrighed is used in many production environments, the biggest clusters (according to project official statistics [19]) contain 400-450 CPU cores.

However, the last project activity is a release of version 3.0.0. in 2010.

2.2.3 XtreemOS

The project started in June 2006 and was funded as an Integrated Project by the European Commission.

2.2.3.1 Architecture

XtreemOS [20] is a Linux operating system based on OpenSUSE. The aim of the project is to create a planetary-scale computing platform to support virtual organizations to allow resource federation. The system aims more at scientific computing rather than to be a general-purpose cluster. Unlike Clondike, where all nodes are equal, XtreeemOS has a Core node and multiple Resource nodes.

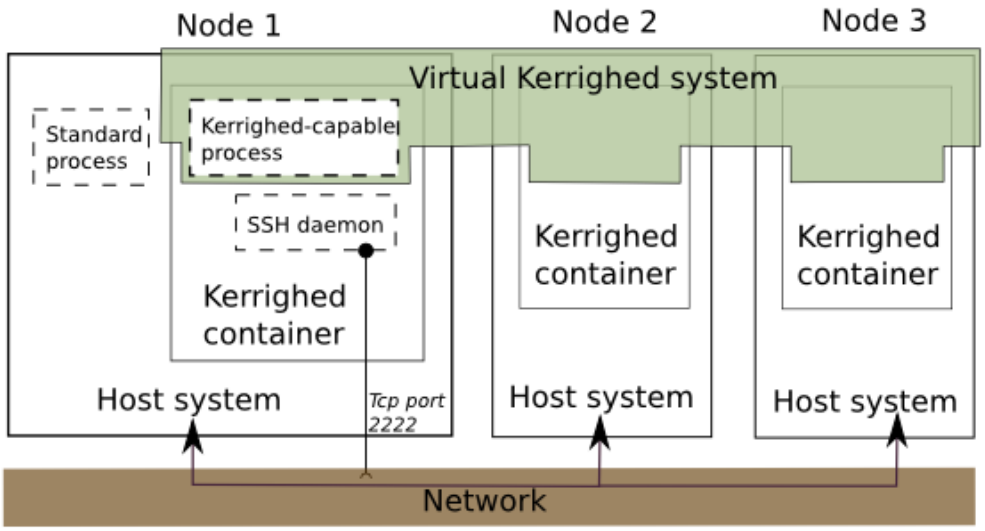

Figure 2.4: The Kerrighed architecture

2.2.3.2 Current state

XtreemOS 3 was released in 2012, the system is still being developed. However, most activities ended with the end of the EU funding in 2010. XtreemOS team does not provide any information about real users of the system, so we cannot qualify if it is only an academic project or if it has some footprint in a real production environment as well.

2.2.4 Condor

Condor [21] project is being developed at the University of Wisconsin-Madison and is freely available under the Apache License.

Condor is a specialized workload management system for compute-intensive jobs. It utilizes waste computing power of idle workstations.

Condor provides job queueing mechanism, scheduling policy, priority scheme, resource monitoring, and resource management. The main difference to Clondike is that Condor requires a job submission, whereas Clondike is transparent to a user. Every job to submit must have a description file where is specified its architecture, log files and program arguments.

There are two types of jobs - a `vanilla universe`, where a job is run on a remote node as it is and a `standard universe`, where a target application is relinked with the Condor I/O library which provides remote job I/O and job checkpointing.

A job is assigned to a workstation whenever the workstation becomes idle. It sends a request to a Condor server for a job. As the workstation becomes used, it checkpoints the

job and returns it to a job queue. After reassignment of the job to another workstation the job is resumed where it was interrupted.

2.2.4.1 Architecture

Condor is implemented on a library level, so it doesn't need any kernel modifications unlike Clondike. Nodes in Condor cluster have different roles. Basic nodes are Central manager, Execute node and Submit node. Central manager is a head node that controls jobs assignment. This is in contrast to Clondike, where scheduling is on a peer-to-peer basis. Execute node is a node that runs the code, Submit node can submit jobs. Like in Clondike, a submit node can be any node of the cluster.

2.2.5 Dead clustering projects

OpenMosix was a very promising SSI Linux cluster, but it is dead since March 1, 2008. Another fork continued as the OpenSSI project, but it was renamed to LinuxPMI [22].

LinuxPMI did not declare the end of the project, but their website [22] is unavailable and there are no significant signs of activity in the last year.

2.2.6 OpenNebula

OpenNebula is an open-source cloud management toolkit that can abstract management of different virtual infrastructures.

Grids (like Clondike) offer more fine-grained migration based on system processes, Clouds (like OpenNebula) migrate whole virtual machines (a running instance of an operating system). However, even the abstraction of running tasks is different. A closer look at the OpenNebula's scheduler is inspiring in designing a Clondike's scheduler.

2.2.6.1 Architecture

The OpenNebula manager runs on one node of the cluster as a regular Unix daemon. Each running node has an OpenNebula agent that listens to commands from the OpenNebula manager. Commands are sent using ssh as system commands. OpenNebula supports multiple hypervisors - Xen, KVM, VMware, and multiple interfaces to public clouds - Amazon EC2 Query, Open Cloud Computing Interface, and vCloud.

Every node can run its own OpenNebula manager and form its own cloud, which makes it similar to Clondike. Grid systems discussed earlier form only one grid.

OpenNebula offers a unified form of configuration of virtual machine images, virtual machine instances and virtual network infrastructure to a user. Moreover, OpenNebula provides its own API, so the configuration and commands can be run from the 3rd party software.

With the ability of live virtual machine migration, the cloud approach has to solve the same issues as the grid approach - load balancing and high availability.

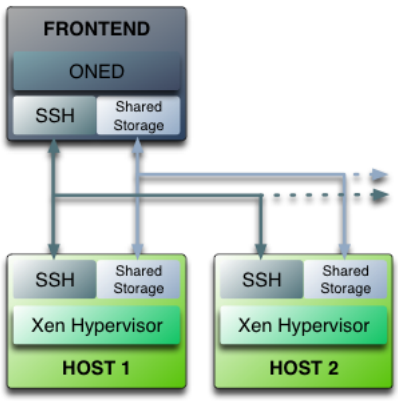

Figure 2.5: OpenNebula architecture

2.2.6.2 Schedulers

OpenNebula provides a scheduler module that is in charge of the assignment of pending virtual machines between known cloud nodes. The OpenNebula scheduling framework is designed in a generic way, so it is highly modifiable and can implement different scheduling policies. Some of them are inspiring for the Clondike's scheduler.

The Match-making Scheduler implements the Rank Scheduling Policy. The goal of this policy is to prioritize those resources more suitable for the virtual machine.

The match-making algorithm works as follows:

- First, those nodes that do not meet the virtual machine requirements (e.g., do not have enough memory, CPU cores, or use another architecture) or do not have enough resources available are filtered out.
- The placement policy is evaluated using the information gathered by monitor drivers placed on every virtual machine.
- Those resources with a higher rank are used first to allocate VMs.

There are 3 basic placement policies: Packing Policy, Striping Policy, and Load-aware Policy. The packing policy tries to minimize the number of cluster nodes in use. When a new virtual machine should be placed, it uses nodes with more running virtual machines first. The striping policy does the exact opposite, it tries to maximize the resources available to a virtual machine in a node. When a new request on virtual machine placement appears, it uses the node with less running virtual machines first. The load-aware policy tries to maximize the resources available to a virtual machine in a node as well, but by looking at real CPU usage of single nodes. It uses the node with more free CPUs first.

Another 3rd party scheduler for OpenNebula is called **H**aizea [4]. It is a resource manager that can allocate requested resources (e.g., 2 nodes, both with 2 CPUs and 2 GB

Figure 2.6: Haizea lease manager [4]

RAM) in requested time. The main abstraction in resource allocation is called the *lease*. The lease simulates a contract between the resource provider (cloud) and a requester. By introducing leases, OpenNebula cloud achieves much higher efficiency, as seen in Figure 2.6.

There are three interesting lease types:

- Best-effort leases that will wait in a queue until resources become available.
- Advance reservation leases that must start at a specific time.
- Immediate leases that must start right now, or not at all.

In Clondike it only makes sense to think about Immediate leases, but the lease mechanism itself could be very inspiring.

2.2.7 Task-specific computation networks

There are many projects whose purpose is to find a solution to a specific problem using computing power of volunteers. Clondike shares a common feature with these projects, as

all use non-dedicated computing nodes. But the architecture is different, there is only one node that submits jobs and a central scheduler that is in charge of dividing jobs to single computing nodes.

2.2.7.1 SETI@home

SETI@home was the first Internet-based public volunteer computing project. The first goal of the project was to prove a concept of volunteer computing and to optimize distributed algorithms for such an environment. This aim was accomplished.

The computing task is to search for five types of signals in noise coming from the space, that could be signs of extra terrestrial intelligence.

2.2.7.2 Folding@home

Folding@home is a project similar to SETI@home, but its purpose is to simulate protein folding. Protein folding is a key computation task in research into Alzheimer's disease, Huntington's disease, and many forms of cancer, among other diseases.

Folding@home is the first internet-based public volunteer computing project, that uses GPUs, PlayStation 3s and has support for multicore computing nodes. Multicore support is achieved using Message Passing Interface.

Currently a computational performance of the cluster is nearly equal to all distributed computing projects under BOINC (Berkeley Open Infrastructure for Network Computing) combined.

In November 2011 the cluster achieved a performance level of 6 petaFLOPS. The fastest supercomputer K-Computer had in the same time period a performance level of 8.162 petaFLOPs.

2.2.8 Blockchain networks

With the massive grow of cryptocurrencies such as Bitcoin [1] a whole new wave of so-called feature-coins arised. As Bitcoin parity with gold was reached in March 2017 [23] Bitcoin will act more like investment currency not suitable for daily exchange. This frees position for feature-coins. There already exist projects that aim to introduce a peer-to-peer marketplace for storage (Sia, STORJ [24], filecoin), total market capitalization of these three by the time of writing this text is $326,546,064.

There also exist projects that aim to introduce a marketplace for computing resources. This is a different approach to Clondike, where the goal is sharing and fair exchange, not capitalization of resources. However the development in this field since 2016 is exponential and this section introduces the main players.

2.2.8.1 Gridcoin

Motivation of Gridcoin project is to help scientific research by funding volunteer computing nodes. Single projects are working toward a wide range of goals, such as disease research

and eradication (Cancer, AIDS, Ebola, Malaria, and others), mapping the Milkyway, and improving our understanding of mathematics [25].

Architecture of Gridcoin network is decentralized. Gridcoin is an open-source cryptocurrency that works with no central authority. The computing network is called BOINC and utilizes GPU, CPU and sensors for off-chain computations. Single tasks must be submitted to the network on a library or an application level. The BOINC network is currently made up of approximately 500k active users and 4 million registered users.

2.2.8.2 Golem

Golem has similar goals like Clondike - to create a decentralized supercomputer. Golem creates a marketplace of computing resources. The object of migration is in this case a microservice [26] represented by a Docker [27] container. Docker virtualization layer isolates application code from a host system. Unlike Gridcoin it can run general-purpose tasks, the main focus is on internet services.

Golem network doesn't emit own cryptocurrency but uses Ethereum [28] instead. Single payments are realised with Golem Network Token (GNT). Applications are submitted to Application Registry as an Ethereum smart contract. Single applications are sandboxed but on top of that are whitelisted or blacklisted by other users. Each user can use its own whitelist and blacklist.

Computations are not realised on Ethereum blockchain but off-chain and Ethereum blockchain servers as a service unit.

2.2.8.3 iEx.ex

iEx.ex has a strong research background. It relies on XtremWeb-HEP [29] a mature open-source desktop grid software developed at the IRNA and CNRS institutes since 2000. This software is used for off-chain computations, a service layer is maintained by Ethereum smart contracts.

The main focus of the project are HPC and Big Data applications. Enterprise features as fault-tolerance and rollbacks are already implemented in XtremWeb-HEP.

2.2.9 Conclusions

The main difference between Clondike and other distributed computing systems, such as Condor [21] or Kerrighed [18], is the P2P architecture and the single unit of migration. Clondike operates on the operating system level and migrates unmodified system processes between its nodes and therefore is fully transparent to a user, unlike systems that operate on a library level and require a specific job-submission framework like CRIU [30]. Since every computer in a Clondike system forms its own administrative domain, the Clondike system goes beyond boundaries of traditional clustering systems. It is similar to the architecture of P2P-based grid systems.

Chapter 3

Clondike performance scalability in homogeneous environment

3.1 Introduction

To evaluate the performance of Vanilla version of the Clondike cluster, we decided to test it on a complex task. We wanted to verify these results on the same task, but using completely another approach. A suitable testing task for our purposes was chosen a distributed source compilation.

A source code compilation is a non-trivial task that requires many computing resources. As the corresponding software project grows, its build time increases and debugging on a single computer becomes more and more time consuming task. An obvious solution would be a dedicated cluster acting as a build farm, where developers can send their requests. But in most cases, this solution has a very low utilization of available computing resources which makes it very ineffective.

Therefore, non-dedicated clusters is a better platform to perform distributed compilation, where we could use users' computers as nodes of a build farm.

In this chapter, we compare Clondike with `distcc`, which is an open-source program to distribute compilation of C/C++ code between several computers on a network.

A very complex task able to test deeply both systems is a distributed compilation of a Linux Kernel with many config options. We have run this task on a cluster with up to 20 identical computers and have measured computing times and CPU loads. In this measurement, we will present the results of this experiment that indicate the scalability and utilization of given resources in both systems. We also discuss the penalty of a generic solution over a task-specific one.

Results of work discussed in this chapter were presented at IEEE International Parallel Distributed Processing Symposium (IPDPS) Workshops in Shanghai and published in conference proceedings [31].

3.2 Platform Architectures

We compare two approaches to distributed compilation: the `distcc` program [8] running on a network of computers and Clondike. Both systems do not require dedicated nodes or synchronized clocks. They also do not require libraries and header files on server nodes for linking. Both systems provide shared resources to users, e.g., every user can benefit from shared computing performance and run jobs in parallel. Both systems do not have a single point of failure, as they do not have any master node and any central job queue. Scheduling of jobs and job placement policy is done by a client, as both systems have no information about the global state of running jobs.

3.2.1 Distcc

`Distcc` [8] is a program to distribute compilation of a C-language family code among several machines. It is provided under the GNU General Public License. It consists of a client and a server (both can be running on a single-CPU computer). If a client receives a new user job that could be distributed, it chooses servers to send the corresponding tasks to. [1]

`Distcc` in this version is often called **p**lain mode distcc.

3.2.1.1 Distribution of tasks

In `Distcc`, only a part of the whole build process can be distributed. The source code is first processed by a `cpp` preprocessor on the local node. This does not require to distribute header files. After a single preprocessed output file is produced, the file can be compiled locally or can be sent over the network to a remote server to compile it.

`Distcc` uses a very simple load-balancing. Every client keeps a status file with information about running jobs and rejections from servers. Each server has a limited number of open connections with clients. The default value is 4.

3.2.1.2 `Distcc` pump mode

Nils Klarlund, a research scientist from Google Build Tools Team, has extended `distcc` by an algorithm called *pump mode* [32]. This algorithm is able to find out what files will be needed for the preprocessing phase and to distribute the prepocessing to server nodes, so that the client node will not get overloaded with preprocessing.

3.2.2 Distribution of tasks in Clondike

Clondike is based on process migration transparent to a user. For the purpose of the experiments described in this measurement, we have used only the *non-preemptive process*

[1] `Distcc` keeps its own terminology of slave-master roles, slave nodes are called *volunteers*.

migration. End-user transparency is achieved in Clondike by implementation into Linux kernel.

A migration of a process begins when the local migration strategy module selects a migration target. The local process creates a new connection to the chosen target node. The Clondike system on the new target node creates a new process that receives process connection from the home node. After establishing a connection, the home process state is transferred to the target node (open files and environment) and the home process becomes a *shadow process*.

The shadow process forwards system calls from the home node to the migrated process and vice versa and the migration is fully transparent.

This approach allows to distribute compilation preprocessing and linking, in contrast to `distcc`. But Clondike cannot optimize migration decisions based on a Makefile (e.g., to perform the compilation locally if the source code is too small), which `distcc` can.

3.3 Experimental environment

3.3.1 Testing job

There are various SW projects whose build times on a single-CPU computer exceed a tolerable limit. The best-known large project in the open-source community is the Linux Kernel. We have chosen this project for testing both approaches to distributed compilation. Samba or Glibc, as other examples of large SW projects, have smaller build times, order of minutes on modern single-CPU computers. One **t**esting job (one Linux Kernel compilation) consists of hundreds of single **c**ompilation tasks.

We have chosen an older 2.6.32.5 kernel [33] with many enabled options [34]. All options are compiled into the kernel (not as loadable modules) to minimize the linking phase time complexity. We do not build any modules, we invoke command `make vmlinux` to generate a kernel image. Of course, we execute the `make clean` command before each compilation (and we do not take its time into consideration).

3.3.2 Measured parameters

The running time of the testing job is measured with the `time` command. We consider the `real` time output of this command.

CPU loads are measured differently for each system. Clondike has its own monitoring framework, whereas `distcc` measurements use the `uptime` command to capture the CPU load. In Clondike, a system with n nodes equals to n servers, where one of servers is a client as well. In `distcc`, a system with n nodes equals to n servers plus one extra client node (the reason is that the client node in the `disctcc` plain mode is overloaded with preprocessing and running a server on this node would deteriorate the results.

3.3.3 Performance tuning

Both systems work differently and our goal is to optimize each of them to get the best possible performance within the given system configuration. The main parameter of the `make` command is the number of compilation tasks (`make` jobs in the terminology of the `make` command) executable in parallel by a client and defined by switch `make -jX`. Therefore, we have run several tests on each system to find the best possible number of `make` jobs.

3.3.4 Experimental platform

All tests were run at the Czech Technical University in Prague - Faculty of Information Technology. We used a lab of 24 identical computers interconnected with a 100 Mbit Ethernet switch into a star topology. CPUs were Intel Pentium Processor G6950 (2 cores, 3M Cache, 2.80 GHz) with 4GB DDR2 memory and one 7200rpm SATA HDD.

3.4 `Distcc` experimental results

The first test was to measure an time overhead of `distcc`, where 1 `distcc` client is running with the local `distcc` server turned off and with one remote server attached, with respect to the time of the standard `cc` job on a single node. Both tests were performed with hot caches and the values of the measured parameters are an average of three measurements. Results are demonstrated in Figure 3.1.

When we have only one `make` job, `distcc` is slower than `cc`. The slowdown is caused mainly by the network transfer, since individual compilation tasks are executed sequentially on the remote server. If the number of `make` jobs increases, the time decreases. If it exceeds the number of cores of the attached server, `distcc` starts benefiting from the preprocessing on a client node. A delay caused by a network decreases as well, since jobs can be "on-the-way", and when a core finishes its job, it does not need to wait for a network to deliver a new job. From the `make` job count 4, `distcc` does not speedup and a single-node compiler has worse results than for `make` job count 2. The observations indicate that this is due to extra overhead with forking to processes that must wait due to overloaded cores anyway.

3.4.1 Scalability

The following test was to measure the scalability of `distcc`. Since we wanted to find out the best setup to be later compared with Clondike, we executed every testing job with 1 to 6 `make` jobs per node. Our assumption from the previous single-node test was that an optimal number of `make` jobs is between 4 and 6. The extra client did only preprocessing and distribution of jobs, with no server running on it.

In Figure 3.2, we demonstrate an overall view of averages of 3 measurement series with hot caches. We define the speedup S of the testing job as the ratio of the sequential

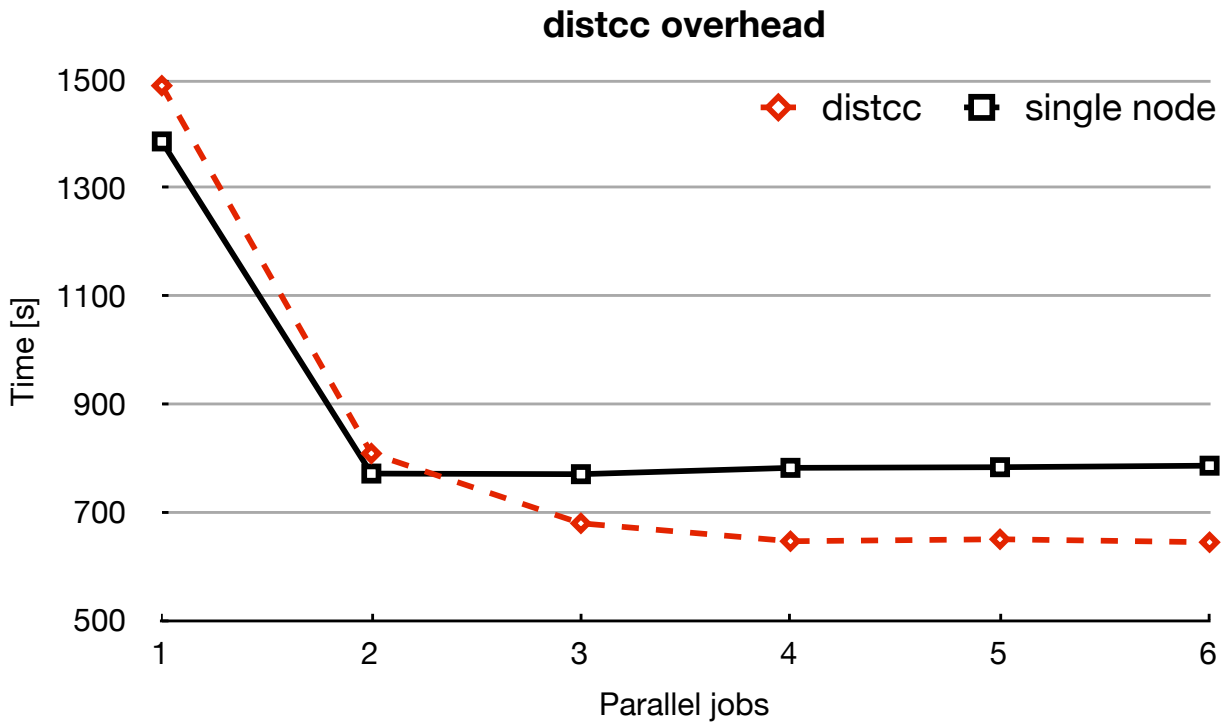

Figure 3.1: Distcc with one server vs a single node.

running time on a single CPU node and the time achieved with p servers:

$$S(n,p) = \frac{T_0(n)}{T(n,p)}$$

We can see that with 2 servers, distcc has a speedup of 1.88 with one make job, 1.74 for with 2 make jobs, and about 1.7 with four and more make jobs. This is a good result - it gives efficiency (S/p) of all server nodes between 84 and 94%. The speedup with multi-make jobs stops for 6 servers at the value of 2.76, keeping the average efficiency of server nodes at 46%. Single-make job measurement copies this trend, the speedup stops growing for 12 servers. This is caused by the overloading of the preprocessing client node.

We have verified one conclusion of the distcc team [8]: An ideal number of make jobs per node is the number of cores plus 2 (in our case of 2-core CPUs, it is therefore 4). More make jobs per node reduce the efficiency and worsen the speedup. If we scale up to 20 nodes with 6 make jobs per node, the time gets worse than with 10 nodes.

3.4.2 Load of individual nodes

We have collected loads of all nodes during the measurement. By the load, we mean the Linux minute load average captured by the uptime command. This load average is

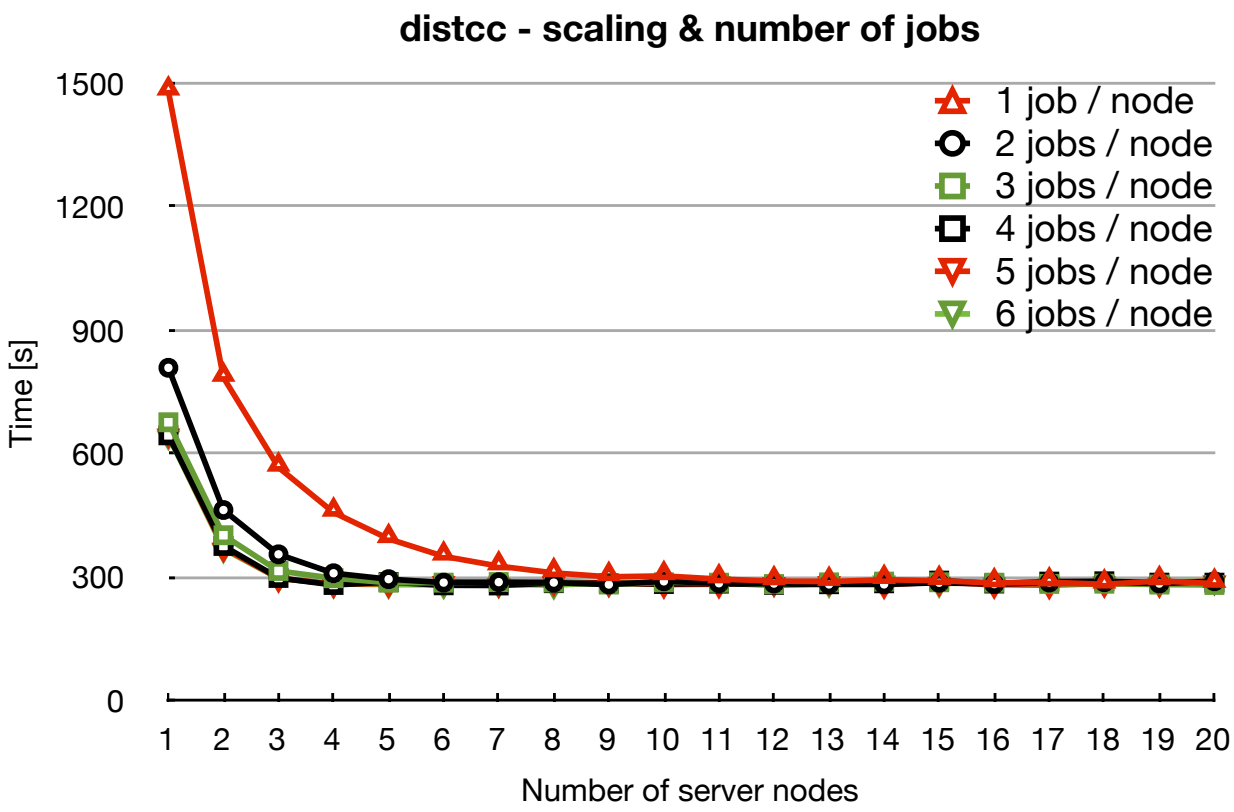

Figure 3.2: `Distcc` on 1-20 nodes with 1 to 6 `make` jobs per node.

described with an analogy of a traffic problem. If we have a bridge, load 0 means that there is no traffic, load 1 means that the bridge is exactly on its capacity, and load 2 means that the waiting queue has the length of the bridge. So, the most common definition of the load is based on the run-queue length: the load is the average sum of the number of processes waiting in the run-queue plus the number of currently executing processes [35].

We can now show how the loads of a client and a server differ. Ideally, a client load should be nearly zero and all servers should have an identical load. One **m**easurement series consisted of running a testing job on 1 to 20 server nodes and for 1 to 6 `make` jobs per server. The testing job was run as a batch, so there was no delay in running particular tasks. A negligible delay in the experimental process was caused by a client running `make clean` after each testing job.

By having testing jobs running in the batch mode, we can easily measure and visualize the load of individual nodes continuously in time, since we know that the number of server nodes is increasing in time. We keep adding nodes one after another, deterministically, there is no randomization, so the first server is running from the beginning to the end of the measurement of the batch job.

In Figure 3.3, we show the load of the client and the first server in 1 measurement

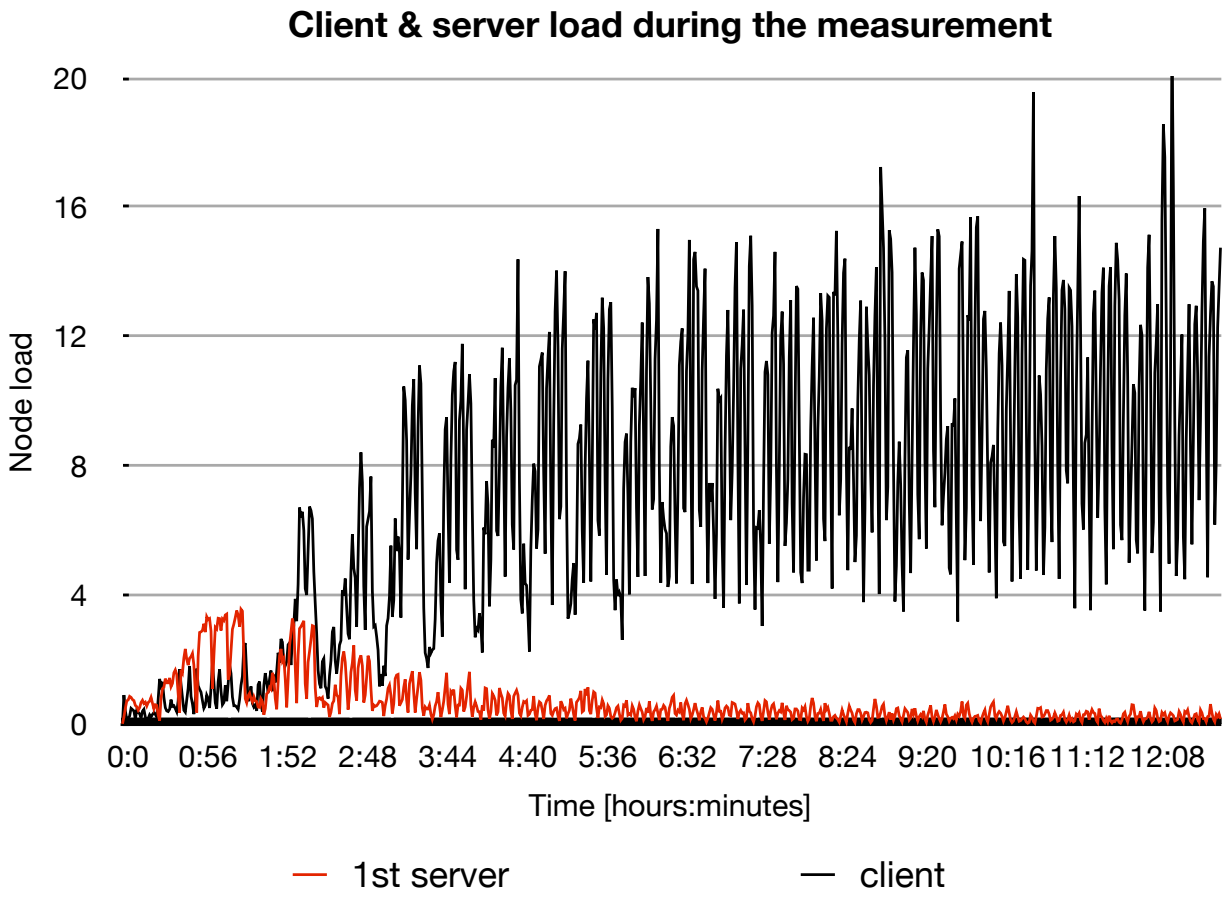

Figure 3.3: Load of the `distcc` client and the first server during the second measurement series.

series. We have run this measurement 3 times (3 measurement series in total), the total time varies from 760 to 770 minutes. This gives us a maximum difference of 2%. However, if we wanted to show an average load in time, we could not use the visualization in Figure 3.3, since the three measurements are close but not equal each to other and the graph would be blurred. Therefore, in Figure 3.3, we show the data from the the second measurement series.

The run of the testing job with 1 to 6 `make` jobs with one server takes 1:21 (1h21m). The server node has a maximal load of 3.55, the maximal load of the client node is slightly below 2, so the two-core node is not still overloaded. If we add one more server node, the load of both servers decreases to 3 (Figure 3.3 shows only the first one). At this point, the client node becomes overloaded, the load exceeds 6 at some moments. As Figure 3.2 shows, we have still a speedup of 1.74. This is also the maximum number of nodes where both servers and the client are overloaded. With 3 servers (time about 2:48 in Figure 3.3), the load of the servers decreases to 2, i.e., they are still well utilized. From 5 servers on, the

load of the client reaches the value of 12 and the efficiency of servers is at 62%. As Figure 3.3 shows, it is the last beneficial enlargement of the network (for 2 or more `make` jobs). From the moment of adding the 6th server (time 3:44 at the time axis of 3.3), efficiency (speedup per server) decreases, new underloaded servers only generate higher load of the overloaded client.

We can conclude from Figures 3.2 and 3.3 that `distcc` scales well up to 5 servers.

3.5 `Distcc` pump results

The `distcc` pump mode authors have released a benchmark showing very promising scalability of this extension [36]. But it turned out that it is quite hard to compile Linux kernels. Finally, we gave up to compile the 2.6.32.5 Linux kernel (used as our testing job). The main problem was that the Linux kernel distributed compilation with `distcc` pump mode requires rewriting of many header files. According to the documentation, this should be overcome by telling `distcc` to reset its caches when the specified rewritten file is changed (parameter `include_server_args` set to `--stat_reset_triggers=file`). But in our case, `distcc-pump` was still failing at the beginning of the build process, switched itself back to the plain mode, and generated an error message `Remote compilation of 'kernel/bounds.c' failed, retrying locally` (i.e., it switched to the plain mode).

So, our experience is that an incremental static analysis of source codes like Linux kernels by `distcc` pump mode is a complicated task and therefore, that the `distcc` pump mode is not compatible with all SW projects. As mentioned in [32], a modification of the Makefile could speed up the pump mode even more, but this is far beyond the scope of this paper. We are looking for a general distributed solution that does not require **a**ny modification of source codes.

In case we would be able to solve issues mentioned before, the difference in scalability between the `distcc` plain and pump modes remains an interesting question.

3.5.1 Modified testing job

As mentioned in [32] and [36], the Samba project (an open-source implementation of the SMB/CIFS protocol) speeded up the build process with `distcc` pump mode twice in comparison with the `distcc` plain mode. So we have decided to compare `distcc` plain and pump mode build speeds of Samba 3.5.9 [37] in our lab. The build of Samba 3.5.9 will be called a testing job in this section.

We have run the Samba testing job on 1 to 20 servers with 1 to 6 `make` jobs per server, using the `distcc` plain mode and then the `distcc` pump mode. This whole measurement series was repeated 3 times and this is called the **b**atch job. The following pseudo-code illustrates its control structure.

```
for 1 to 3 do # iterations
    for plain, pump do # mode selection
```

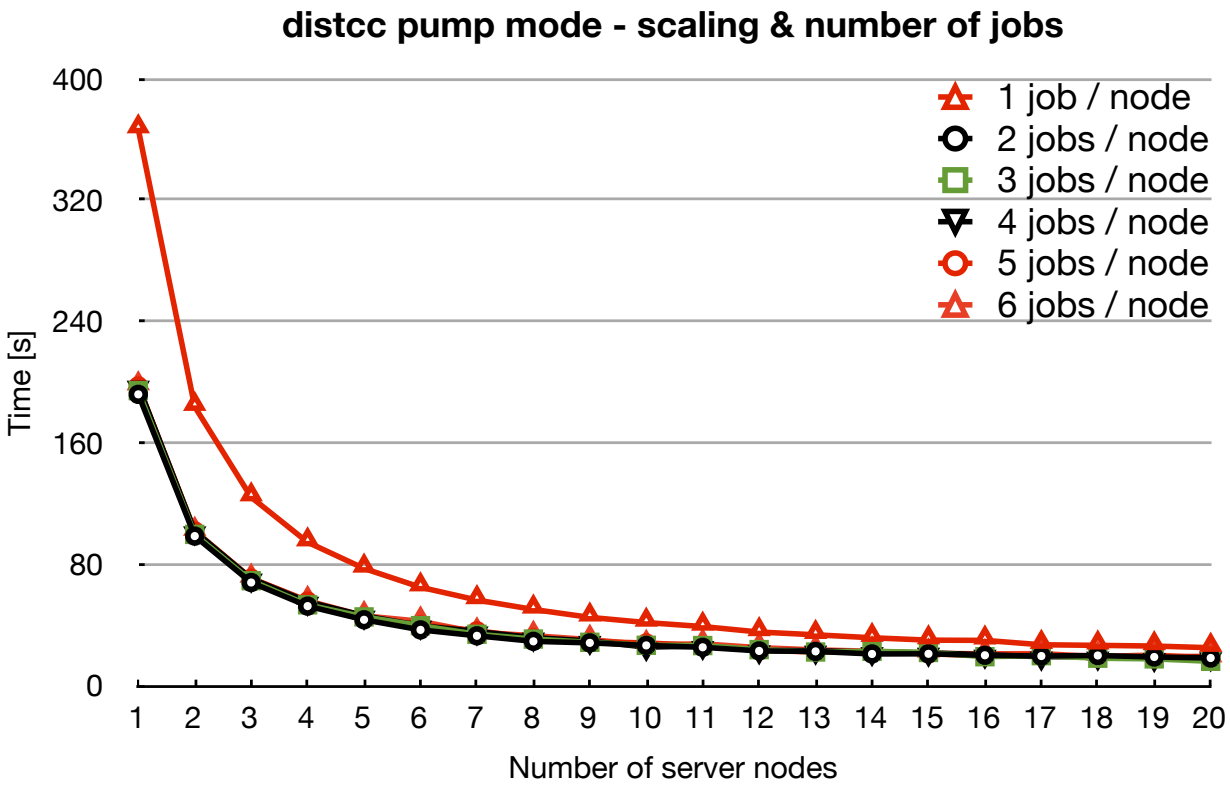

Figure 3.4: Distcc pump mode with 1-20 nodes.

```
        for 1 to 20 do # nodes
            for 1 to 6 do # make jobs
                make clean
                time make
            end for # end make jobs
        end for # end nodes
    end for # end mode selection
end for # end iterations
```

3.5.2 Scalability

In Figure 3.4, we show the same measurement as in Figure 3.2. The result is an average of 3 measurement series. As can be seen, all `make` job counts except 1 have similar times. If we do a sum of times over all number of nodes for every `make` job count, we get a minimum with `make` job count of 2. 1 `make` job = 1405s, 2 `make` jobs = 813s, 3 `make` jobs = 825s, 4 `make` jobs = 840s, 5 `make` jobs = 845s, 6 `make` jobs = 855s. That is much slighter difference than in a case of `distcc` plain mode. This would explain a highly overloaded client node in the `distcc` plain mode.

More important is the scalability itself. For 2 servers and two `make` jobs, we get speedup 1.94, giving efficiency of servers 97%. For 6 servers (where the plain mode stopped to speed up), we have still a speedup 5.3 (efficiency 88%). Speedup continues till 20 nodes. With 20 nodes with 2 `make` jobs, we have a speedup 10.61, giving 53% efficiency (still better than 46% on 6 plain mode nodes). This is a very promising result, but we must keep in mind that Samba project is one of projects that have best results with `distcc` pump mode [32].

3.5.3 Load of individual nodes

Figure 3.5 shows the load of the first server and the client during the second measurement series. Both modes are shown, `distcc` plain mode first, `distcc` pump mode next. We show again only the second iteration of the batch job, to avoid blurred graphs.

The total time of `distcc` plain mode measurement series was 228 minutes. For the pump mode, it was only 134 minutes. But more important is the distribution of the load on individual nodes. As we can see in Figure3.5, in the plain mode, the client is overloaded slightly less than in case of a kernel compilation. This can be caused by that fact that Samba build has less small files than Linux kernel build. But the phenomena of an overloaded client node and underutilized servers is the same.

In Figure3.5, the pump mode starts at 3:48. The distributed preprocessing plays a huge role in the client load. We can see that the dual core node was not overloaded during the pump mode measurement. The client node load reached the peak of 2.07, but even with 20 nodes, it stays between 1-1.5. On the other hand, for less than 10 servers, the servers are overloaded. But even with 20 nodes, we have still a load between 1 and 1.5, which is much greater efficiency than in case of the plain mode.

Figure 3.6 shows the load of all nodes for the pump mode only. It demonstrates that the load between server nodes is fairly distributed (lines overlap). The load of the client stays under 1 until the time 1:24, which represents 15 active server nodes.

Fair distribution between nodes is the same as in the plain mode, but the client node is not overloaded anymore.

3.6 Clondike results

Clondike in Vanilla version is a peer-to-peer grid where every node acts as a server and a client. A node can be a part of many parallel relations. If a node initiates a distributed task, it migrates its processes to other nodes and in this role, they act as computing servers for the client node similarly to `distcc`.

Unlike `distcc`, an initiating node (client) does not understand the executed task at all, it just migrates processes using given scheduling rules. In the case of `distcc`, we measured 20 servers + 1 client, see Section 3.3.2. Based on the previous experience with Clondike computations, we have chosen the number of `make` jobs equal to 10 as the best one.

We measure the same testing job in the case of the `distcc` plain mode, i.e., the build of a 2.6.32.5 kernel [33] with the configuration published on the web, see [34].

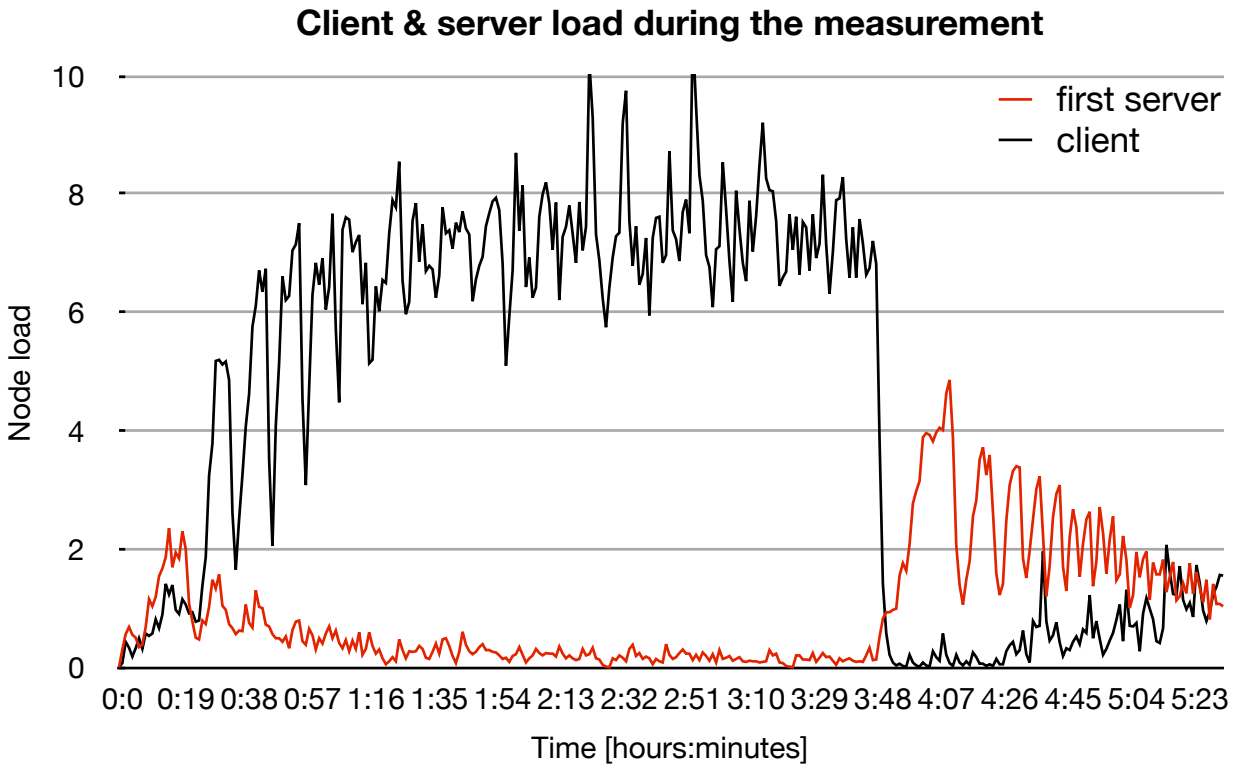

Figure 3.5: Load of the client and the first server in `distcc` plain and pump mode during the second iteration of the batch job.

3.6.1 Clondike vs `distcc`

Figure 3.7 compares both systems using the testing job. Both systems were tested with an optimal number of `make` jobs. As we can see, Clondike with one node has worse time than `distcc` (842s vs 644s). Even Clondike has some internal overhead, this is caused mainly by the fact that `distcc` uses one server, but the client node does all the preprocessing. From two and more servers, `distcc` starts losing this advantage, since its client node becomes overloaded. With 2 nodes, Clondike has speedup 1.67 (83% efficiency), with 4 nodes 2.35 (0.58%). Speedup stops at the 6th node, where Clondike reaches the best absolute time 319s.

3.6.2 Load of individual nodes

Clondike has its own logging framework that samples load every second, so we are able to obtain finer results. Figure 3.8 shows the load of 5 nodes (one of them playing the role of the client) during the testing job. As we can see in Figure 3.7, Clondike with 6 servers does not speedup. Unlike `distcc` where all server nodes (not the client) have almost the

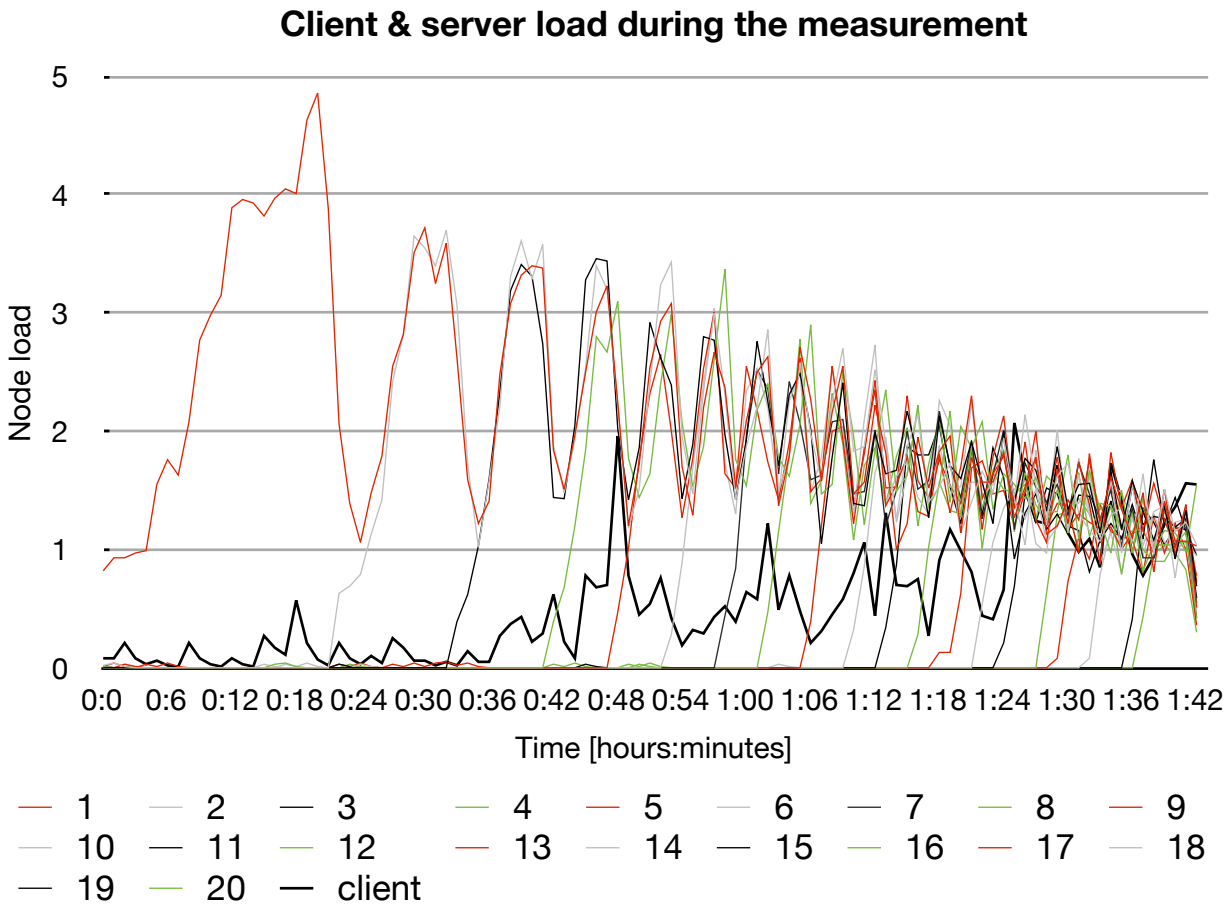

Figure 3.6: Load of the client and all servers in the `distcc` pump mode during the second iteration of the batch job.

same load (see Figure 3.6), the load of Clondike servers differs. But the difference is still acceptable.

3.7 Multiple tasks in parallel

Since the `distcc` plain mode suffers from low efficiency of server nodes, we added one more client node. In this measurement, we have initiated a testing job on two client nodes in parallel (each server has to serve two clients at time).

Figure 3.9 compares Clondike and the `distcc` plain mode with two concurrent testing jobs. `Distcc` speeds up up to 9 server nodes. (Recall that with a single task, the speedup stopped with 4 servers). This is a result of two preprocessing clients who are able to fill more servers with data. With total times 293s and 298s, both kernels are compiled only a

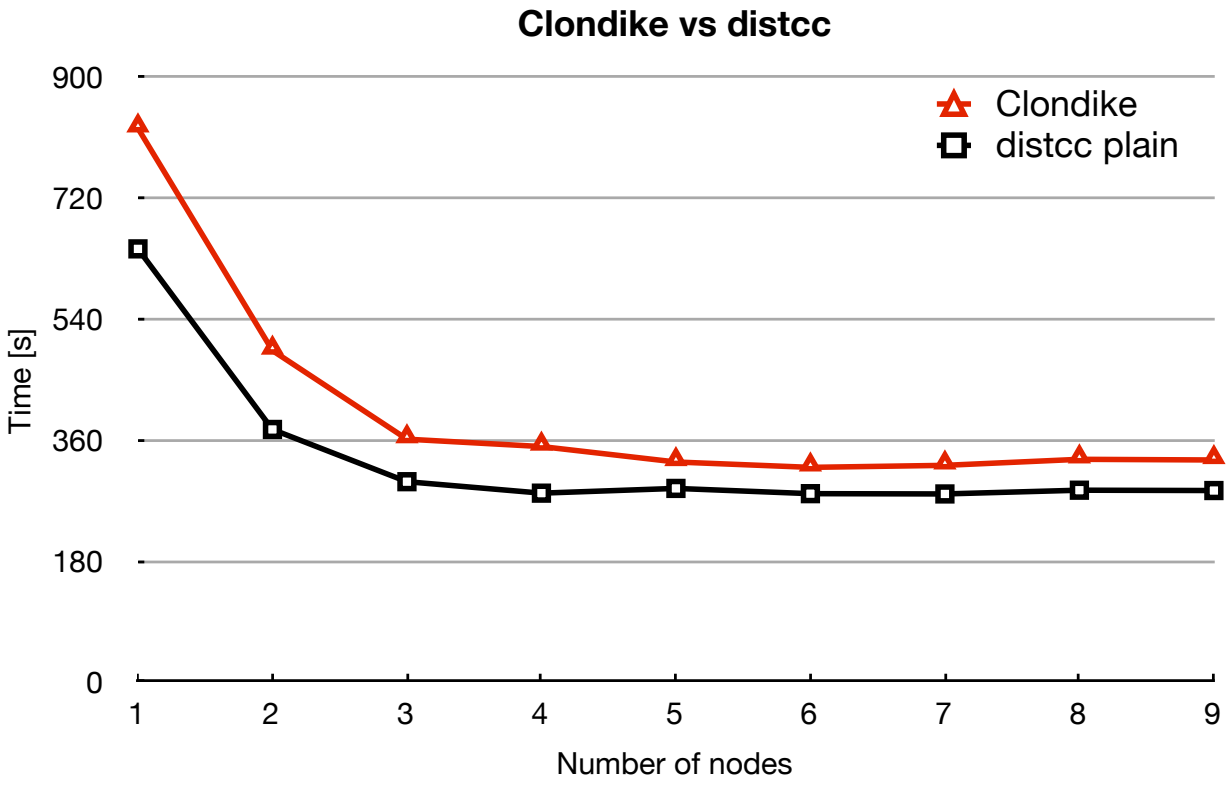

Figure 3.7: Clondike and `distcc` plain mode on 1-9 nodes.

few seconds later than one kernel (281s).

Clondike has a very close time of one testing job to two parallel testing jobs as well. This is caused by caching of Clondike filesystem that in case of similar tasks speeds up the process. The only problem is with 4 servers, where Clondike reaches worse time than with 3 servers. We believe that this is caused by the Clondike process scheduler. As we can see, the total times of both projects are very close. But in this case, it is $n + 2$ servers on the side of `distcc` compared to n servers of Clondike. As Clondike is a general and `distcc` a task-specific cluster, the results of Clondike are very promising.

3.8 Conclusions

We have compared two distributed computing platforms, general-purpose non-dedicated cluster Clondike and special-purpose distributed compiler `distcc`, using a distributed source code compilation benchmarking. `Distcc` was tested in two modes — plain and pump.

`Distcc` in the plain mode was able to scale to up to 5 servers with uniformly distributed load. The limiting factor of its scalability is overloading of the client where all preprocessing

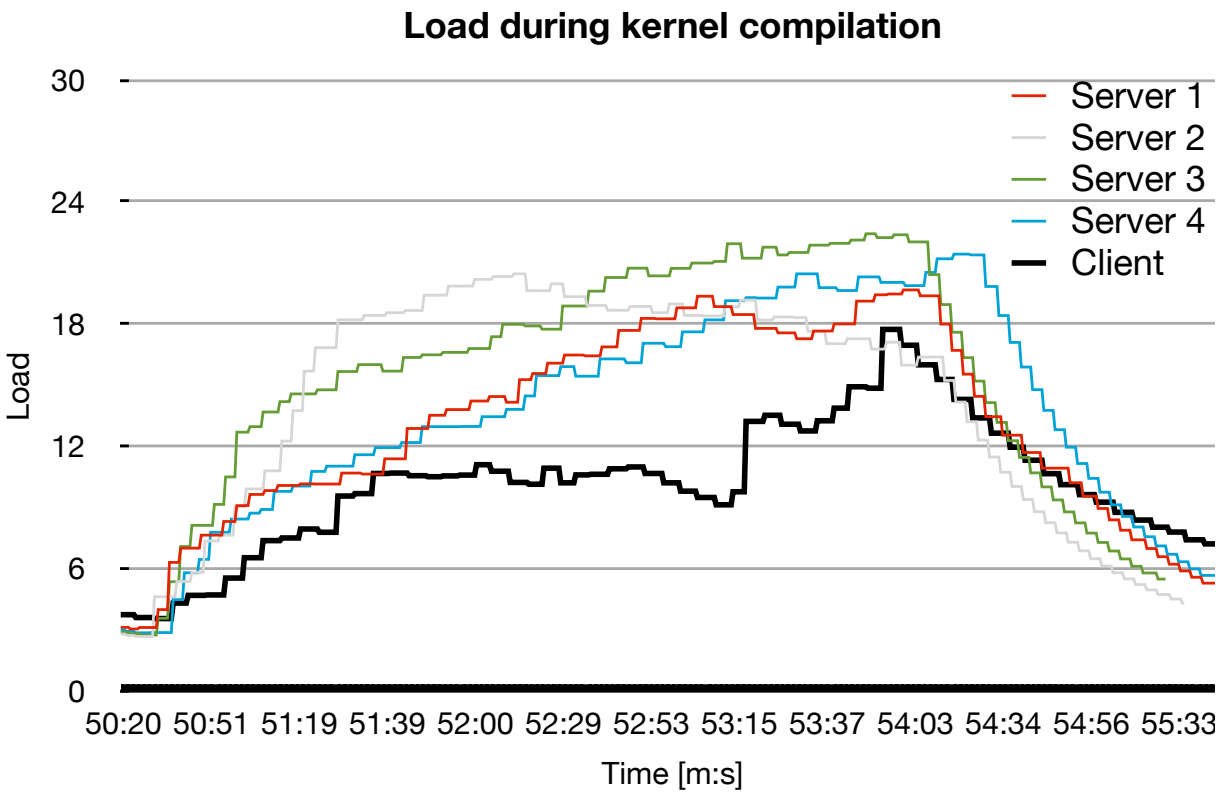

Figure 3.8: Loads of Clondike nodes during the testing job (run on 5 nodes).

takes place.

`Distcc` in the pump mode solves this issue by distributing all phases of the build process among all server nodes. The measurement has confirmed practical scalability up to 20 server nodes. The drawback of the pump mode is that in many SW projects, the `distcc` compiler must be specifically tuned and so it is not an universal solution to distributed source compilation.

Clondike was able to scale to up to 6 computing nodes and was fully competitive with the special purpose `distcc` solution. Since it is an universal clustering solution, all phases of the compilation process are distributed to server nodes. One limiting factor of this universal approach is an impossibility to prevent distribution of small tasks, where the task distribution takes more time than local processing. A process scheduler that could be able to learn and evaluate which tasks to migrate is an subject of further research.

We can conclude that in cases where we look for a distributed compiler able to compile many different projects without tuning and modifying the compilation process, we can replace the `distcc` plain mode with Clondike. We pay just a small performance penalty, but gain a universal cluster solution that allows to run other non-compilation tasks in parallel.

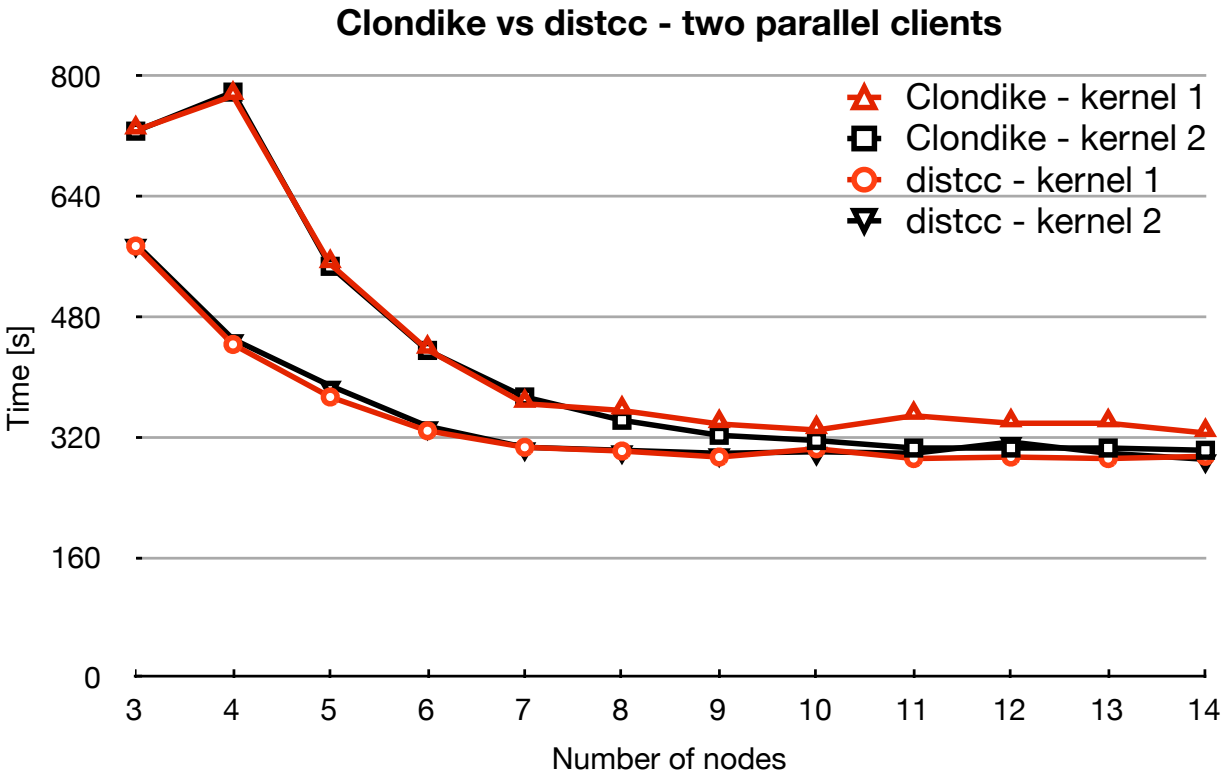

Figure 3.9: Clondike and `distcc` plain mode with 2 parallel testing jobs on 3-14 nodes.

Chapter 4

Clondike performance scalability in heterogeneous environment

4.1 Introduction

Benchmark tests of Vanilla Clondike described in the previous chapter were performed in our homogeneous computer laboratory (all nodes are the same). In this set of measurements, we moved the cluster closer to a real environment. From a homogeneous computer laboratory environment, we ported Clondike to a real office with heterogeneous computer nodes. Next, we have run the same benchmark - a distributed compilation of a Linux Kernel - on both platforms to verify our results and to identify future research goals.

Results of work discussed in this chapter were presented at IEEE Parallel Distributed and Grid Computing (PDGC) conference and published in conference proceedings [38].

4.2 Experimental environment

To verify results of porting, we had to choose an appropriate task that would test the cluster on both platforms.

We have chosen the same testing job as in previous measurements on a homogeneous cluster described in Section 3.3: compilation of the 2.6.32.5 Linux kernel [33] with many enabled options [34].

4.2.1 Experimental platforms

We have compared two platforms. The *homogeneous platform* is the current platform, located at the Czech Technical University in Prague - Faculty of Information Technology, where Clondike runs and where previous measurements were done, see Section 3.3.4. The *heterogeneous platform* is an existing office infrastructure environment and we had to port Clondike to be able to run there.

4.2.1.1 Clondike installation on the homogeneous platform

The computer lab is shared with other projects, so we had to make a mechanism to load Clondike on all computers on demand. This goes against the philosophy of Clondike, Clondike should be used as a non-dedicated cluster, but the kernel patch is not yet stable enough to be integrated in other projects kernels as well.

We have installed one server in the lab that provides a `PXE` [39] boot server for all machines. The default boot option is a local hard drive boot when chosen a `PXE` boot, it boots a temporary in-memory operating system from the server.

We have stored raw images of all operating systems we use in the lab on the server. By setting an option on the server, we specify which image should the temporary operating system copy on the local hard drive of the single machine, that booted this system. After a reboot, this single machine loads the selected operating system from the local hard drive.

One drawback of this approach is that we completely erase the local hard drive every time we want to change an operating system.

If we wanted to store the local operating system of some machine, we choose a specific upload operating system on the server and boot it using `PXE` on the desired machine. This upload operating system makes an image from the local hard drive and copies it to the server over a network.

From now on, we can use this image on all computers.

4.2.1.2 Clondike installation on the heterogeneous platform

Heterogeneous platform is an existing office infrastructure with computers interconnected using 100 Mbit Ethernet switch into a star topology. Configuration of single computers differs, we wanted to have nodes with different computing power. By a **n**ode we denote either a single computer or a virtual machine running an instance of Clondike kernel.

The specific experimental setting was the following:

1. Node #1 is the second most powerful node of the Cluster. It is a MSI notebook.
2. Node #2 is a virtual machine, running in VMware Fusion on OS X. It has allocated 1 core out of 2 cores of the hosting Macbook Pro.
3. Nodes #3, #4, and #5 are standard desktop computers.
4. Node #6 is the most powerful node of the cluster, it is a standard desktop computer.
5. Node #7 is the least powerful node of the Cluster. It is a standard desktop computer.
6. Node #8 is a virtual machine, running in VMware Fusion on OS X. It has allocated 1 core out of 2 cores of the hosting Macbook Pro.

A detailed hardware description of single nodes is in Figure 4.1.

We still didn't want Clondike to run on a production kernel, but the installation mechanism from the homogeneous platform was not reusable.

	CPU	# of cores	# of threads	GB RAM
Node #1	Intel(R) Core(TM) i7-2670QM CPU @ 2.20GHz	4	8	8
Node #2	Intel(R) Core(TM) i5 CPU M 520 @ 2.40GHz	1	1	2
Node #3	Pentium(R) Dual-Core CPU E5200 @ 2.50GHz	2	2	2
Node #4	AMD Athlon(tm) 64 X2 Dual Core Processor 540	2	2	1
Node #5	AMD Phenom(tm) 9350e Quad-Core Processor	4	4	4
Node #6	Intel(R) Core(TM) i7 CPU 870 @ 2.93GHz	4	8	16
Node #7	AMD Sempron(tm) Processor 3400+	1	1	1
Node #8	Intel(R) Core(TM) i5 CPU M 520 @ 2.40GHz	1	1	2
	Complete cluster	**19**	**27**	**46**

Figure 4.1: Hardware specification of nodes of the heterogeneous cluster.

We wanted to have a single kernel instance as well, but we wanted to skip the phase of erasing the local hard drive. Our experience showed us that after a semester (4 months) of daily hard drive reformatting, we lose about 8% of hard drives.

As memory is getting cheap enough and networks fast enough, we decided to make the installation completely diskless. But we wanted to avoid a bottleneck caused by a single server, so a complete installation on a distributed file system was not an option.

We decided to load the operating system over the network from the server using `NFS` [40], but to put as much writes as possible to the main memory using a temporary file system `tmpfs` [41]. The Clondike architecture induces massive writes, since with each migrated process it migrates all files the process needs and stores them on the local node.

But even in a case of a kernel compilation a 1 GB temporary `tmpfs` file system on the computing node can handle this demand. The initiating node has to have 2 GB temporary file system, as our kernel configuration produces 1.6 GB of data. Other writes to the file system are unique, load of the `NFS` server during the whole measurement did not exceed 1%.

Another issue to solve was heterogeneous device drivers. We wanted to have a universal instance of a filesystem we could easily copy for every new node. Previous Clondike installation was tuned to fit the lab, so we had a custom kernel with all drivers compiled in. Now we needed a kernel that would load on (almost) every hardware, so we had to compile all possible device drivers as modules.

The last issue to solve was the boot process itself. We have made a boot menu using `Syslinux` [39] with 8 different Clondike instances. Every computer had to choose one instance. This loaded a `NFS root` [42] enabled kernel with a root partition pointing to one of 8 `NFS` shares.

Single instances did remain hardware-agnostic, so it did not mater which one was loaded on which computer.

With this approach, we successfully migrated Clondike on a heterogeneous platform and extended the possibilities of testing. A big advantage is a completely diskless setup,

so we can use ordinary production computers without touching their hard drive.

4.3 Results on the homogeneous platform

Recall the results from the previous chapter. Figure 3.7 shows Clondike and `distcc` plain mode scalability on the homogeneous platform using the testing job.

The scalability trend of `distcc` copies results of Clondike, so we can assume that Clondike on the homogeneous platform runs fairly optimal.

Figure 3.8 shows the load of 5 nodes (one of them playing the role of the client) during the testing job. The load of single Clondike servers is not much different.

4.4 Results on the heterogeneous platform

We have measured the same testing jog, i.e., the build of a 2.6.32.5 kernel [33] with the configuration published on the web, see [34].

Since the computing power of both platforms is incomparable, we were not interested in total compilation times. We were interested in system scalability on both platforms.

4.4.1 Scalability

Figure 4.2 shows Clondike scalability on the heterogeneous platform using the testing job. We have initiated the testing job on the second fastest computer (`Node #1`). It did run with 16 concurrent `make jobs`, since `Node #1` has 4 cores and 8 threads.

The second added node (`Node #2`) had a weaker hardware, but we still had a speedup 1.26. But after adding the third node (`Node #3`) we did have a negative speedup 0.53, so the testing job ran slower than with only one node.

The reason is that the testing job (compilation of the Linux kernel) consists of many single tasks that are migrated as single processes, but they still have some dependencies. So, if a compilation task is migrated to a slower machine, another task may not be executed before the first task finishes.

The fourth added node (`Node #4`) turned out to be the weakest and did cause a huge slowdown of the whole cluster. The speedup was 0.07. The result is caused by two factors - a combination of dependencies in the Makefile and a round-robin-based scheduler in Clondike. Another factor may be hardware problems of the computer, because it showed weaker results as much older one-core Sempron CPUs.

After adding more nodes, the probability of a single task to end on the slow `Node #4` decreased, so the cluster did speedup again (compared to the worst result with 4 nodes).

The measurement did show us that from the total-time point of view the current Clondike process scheduler did not benefit from more nodes. An upgraded heterogeneous-platform agnostic scheduler is an subject of ongoing research.

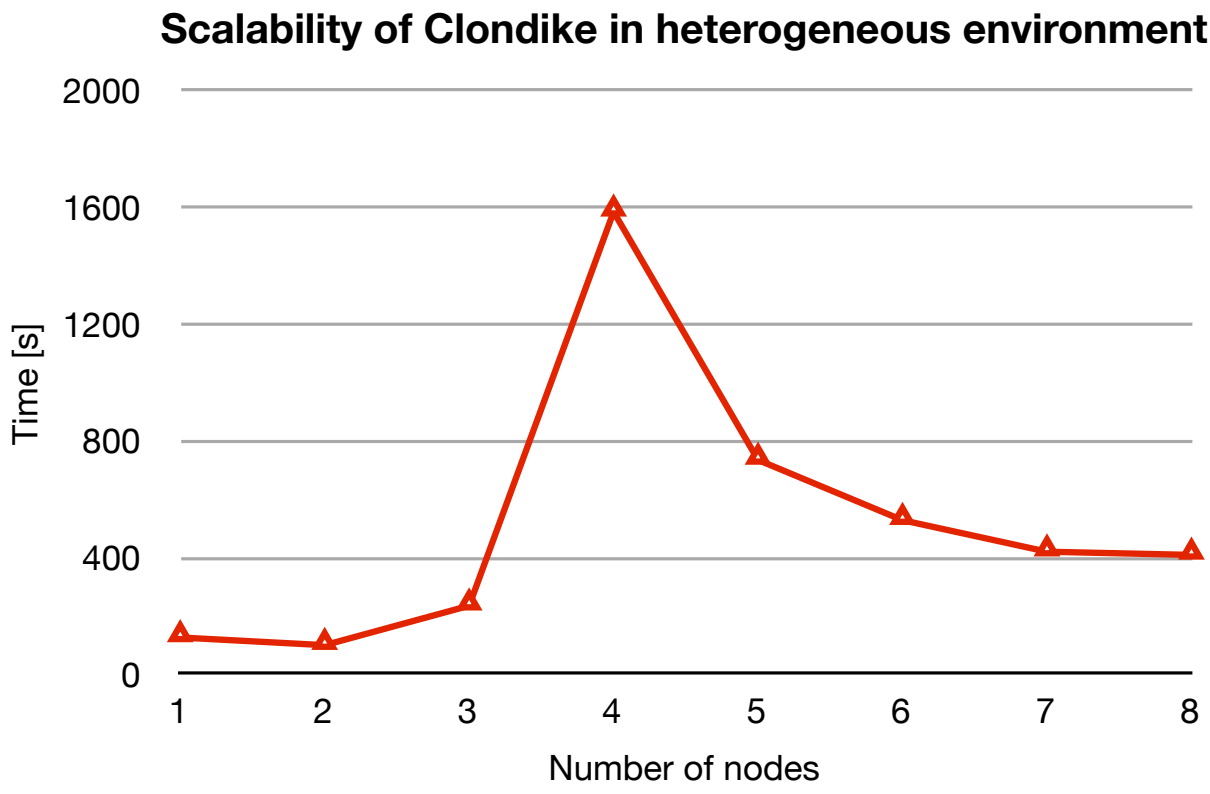

Figure 4.2: Clondike on the heterogeneous platform.

4.4.2 CPU load of individual nodes

Figure 4.3 shows the load of 5 nodes during the testing job on the heterogeneous platform. Unlike in Figure 3.8, where all nodes show relatively the same load, here the loads of individual nodes differ.

Figure 4.3 shows a measurement with 5 nodes, the slowest nodes of this configuration are (`Node #3`) and (`Node #4`). These two are also the most overloaded. The fastest node of this configuration is (`Node #1`) and it is also the most relaxed one.

4.5 Conclusions

We have successfully ported Clondike from a laboratory homogeneous platform to a real office `heterogeneous platform`. From now on we are able to run Clondike on any computer without touching its local hard drive.

To fulfill a requirement on a diskless system, we had to run Clondike from a distributed network file system. This solution required multiple instances of the operating system (one per node). To make these instances hardware-agnostic, we had to change a way Clondike

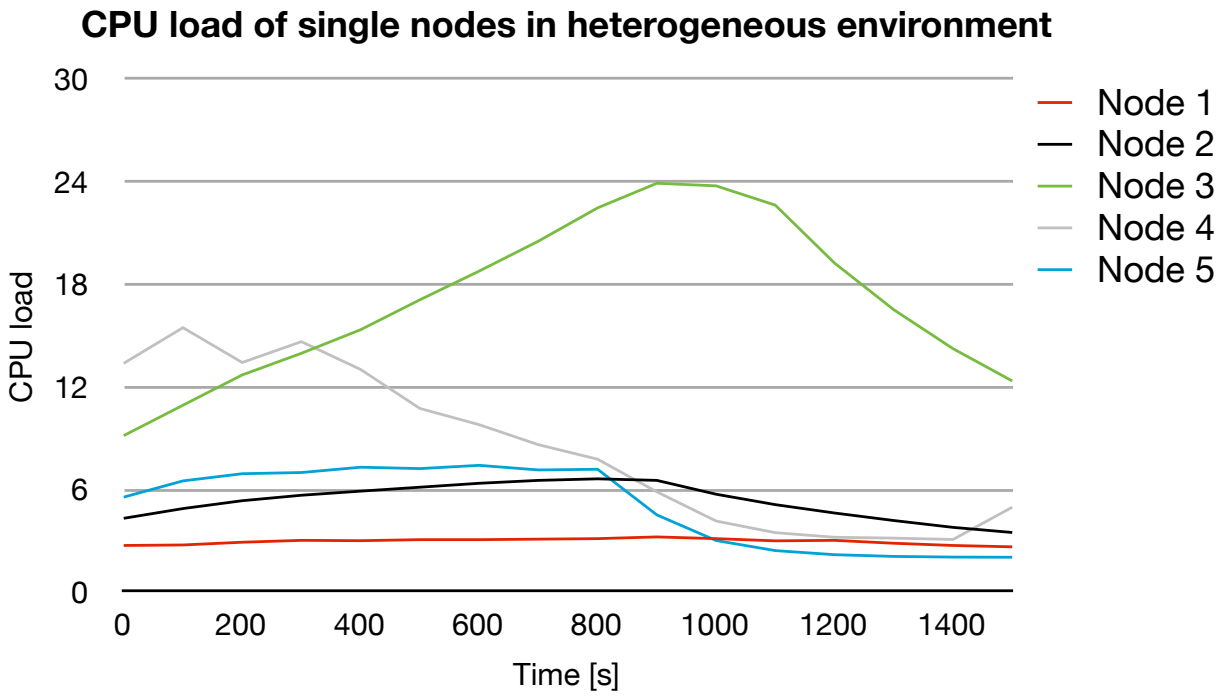

Figure 4.3: Loads of single Clondike nodes on the heterogeneous platform with 5 nodes.

loads device drivers. To overcome a bottleneck of the distributed file system we have implemented an in-memory file system on every Clondike instance and redirected all write requests to the main memory of each node.

We have done a set of measurements to compare both platforms.

Results from a heterogeneous platform showed us current limitations of Clondike. One slow node slowed the whole cluster down. After adding more nodes, Clondike continued decreasing the total time of the testing job until the total number of 8 nodes.

A more sophisticated scheduling algorithm, that would migrate less tasks to less powerful nodes and utilize the computing power of more powerful nodes is an object of future research.

Chapter 5

Scalable decentralized node discovery and bootstrapping system

5.1 Introduction

P2P systems have the ability to scale out [43], but there are technical challenges coming from their zero-master architecture that must be solved for each such system. Two of these challenges are 1) the protocol which individual nodes communicate among themselves with and 2) the initialization protocol which nodes get to know about each other with. In the further text, the initialization protocol will be called the `bootstrapping`.

Recently, the development of the Clondike cluster [11] focused more on performance tuning [31] and on resource planning in different environments [38]. Clondike Vanilla implementation of bootstrapping and inter-node communication was based on broadcasts and therefore, its communication overhead became a limiting factor of scalability of the whole cluster. Therefore, we decided to replace the broadcast by a more sophisticated and scalable P2P protocol and introduce a new version Clondike Chameleon.

Bootstrapping and inter-node communication is solved in many contemporary P2P file-sharing protocols on a very large scale, such as BitTorrent [5]. But Clondike is unique as a general-purpose computing cluster (rather than a data-exchange system only). Therefore, it has some specific requirements that demand a deeper analysis of possible solutions.

In this chapter, we provide an overview of existing inter-node and bootstrap communication methods in P2P systems and choose an appropriate solution for Clondike. We describe the implementation of this solution and verify the results by a series of measurements.

Results of work discussed in this chapter were presented at Parallel, Distributed and Network-based Processing (PDP) conference in St. Petersburg and published in conference proceedings [44].

5.2 Bootstrapping protocol in Clondike

5.2.1 The previous state

In Vanilla implementation of Clondike, the bootstrapping protocol was based on the broadcast operation. Each node sent status messages periodically over the UDP broadcast and each node listened to incoming broadcasts. This solution had two drawbacks. First, the scalability was limited, since the number of messages grows with the number of nodes and all messages were sent periodically. By the *number of messages* we mean individual communication messages of the protocol, not packets on the network layer. Second, the cluster had to be formed on the same subnetwork, since UDP broadcasts do not overcome *NAT* (network address translation). The goal of Clondike Chameleon is to create a global P2P network that is not limited to a single location, e.g., a single subnet.

5.2.2 Requirements

We have defined the following requirements on the new bootstrapping protocol: 1) to minimize the number of periodical updates, 2) to be NAT-traversal friendly, 3) to scale logarithmically, 4) to allow frequent node bootstrapping, 5) to allow frequent node disconnection.

The user space of Clondike (where the bootstrapping is placed) is written in Ruby, so the implementation must be either in Ruby or C/C++. The following functionalities must be implemented:

- (F1) creation of a new node,
- (F2) connecting to the cluster,
- (F3) connecting to a remote node,
- (F4) discovering of IP addresses and ports of known nodes,
- (F5) enforcing of search and force connection to a given node in the cluster (by providing a static link to a connecting node).

5.2.3 Overview of P2P networks

In order to find a suitable bootstrap and inter-node communication protocol for Clondike, the topology of the underlying network must be considered.

P2P systems are classified based on their network topologies as *structured*, *unstructured*, or *hybrid* [45].

5.2.3.1 Unstructured P2P networks.

Unstructured P2P networks are P2P networks of the first generation. They are arranged into random graphs. Individual requests in these networks are propagated either by a recursive flooding algorithm (deterministic networks) or by pseudorandom forwards between neighbours until the target is reached or the number of forwards exceeds a given maximum. Each node has its own address space to identify other nodes. Examples of well-known unstructured P2P networks are FreeNet [46], Gnutella [47], or BitTorrent [48] (in the original version without distributed-hash-tables extension).

5.2.3.2 Structured P2P networks.

Structured P2P networks usually use *distributed hash tables* (DHT) and have rules for creating links between nodes. These networks guarantee that if the requested content exists in the network, it is found. The most-known structured P2P networks are Tapestry [49], Pastry [50], CAN [51], Chord [52], and Kademlia [53].

However, Kademlia is the only network that disconnects nodes in zero communication steps, which we found a key advantage over other alternatives as it allows frequent node disconnection. Therefore, this chapter will further focus only on Kademlia.

5.3 Kademlia

Kademlia is a P2P system developed at the University of New York in 2002 [53]. The main idea is based on a metrics that defines the distance between two nodes x and y as bitwise exclusive or (XOR) of their identifiers interpreted as an integer [53].

$$d(x, y) = x \oplus y$$

Each node keeps links to remote nodes in a *bucket*. A bucket is a structure that keeps links to other nodes from a selected area of node identifiers space. Node links are organised into buckets according to their distance to a target node. Each bucket contains at most k links (and therefore are sometimes called $k - buckets$). k is a redundancy parameter of the network. By design each hop reduces a distance to destination to one half, so any node can be found in $O(\log_2 N)$ steps where N is the number of nodes at the given time. When a new node connects to a network, it generates a random 160 bit *nodeID*, connects to any connected node, and searches for nodes closer to its *nodeID*. It finds its closest neighbours in the node identifiers space in $O(\log_2 N)$ steps. The average number of hops in Kademlia network is logarithmic [53]. One key feature of the Kademlia network is the fact that a node can disconnect itself with zero communication steps because each node sends periodic *keep-alive* messages to nodes in its bucket, so a disconnected node is discovered due to the absence of responses to keep-alive messages.

The first large-scale deployment of Kademlia was in Azureus BitTorrent client [54], where Kademlia was used to extend the current BitTorrent protocol. It superseded Azureus

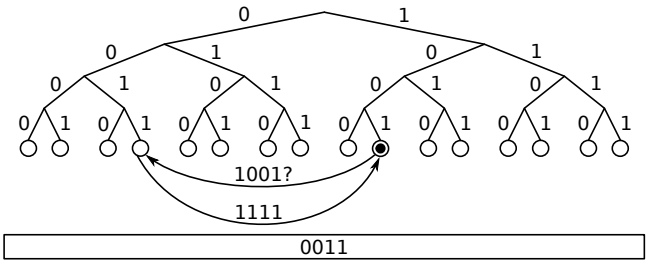

Figure 5.1: New node 1001 to be connected to the system knows only node 0011. 1001 asks 0011 for $k = 1$ link to a node closest to 1001. Node 0011 replies with the link to 1111.

tracker with DHT and therefore, it is referred as the *Azureus DHT*. The arrival of the Azureus DHT influenced the original BitTorrent protocol and an official extension of BitTorrent was created. The extended protocol is referred as the *Mainline DHT* (MLDHT) [9]. Both protocols are based on Kademlia, but they are not compatible with each other. Most of the BitTorrent clients nowadays (including Azureus) implement MLDHT. According to [55], the MLDHT is used by 10 to 25 millions of clients every day.

We present the process of connection of a new node and a routing table in following figures 5.1, 5.2, 5.3, 5.4.

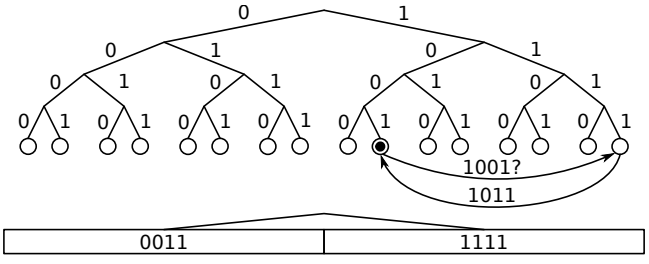

Figure 5.2: Node 1001 adds the link to 1111 and splits its first bucket to two (the capacity is only 1). 1001 asks 1111 for 1 link to a node closest to 1001. 1111 replies with the link to 1011.

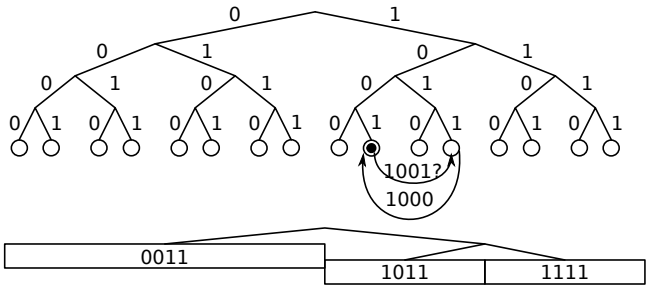

Figure 5.3: Node 1001 adds the link to 1011 and again splits a full bucket. 1001 asks 1011 for 1 link to a node closest to 1001. 1011 replies with the link to 1000.

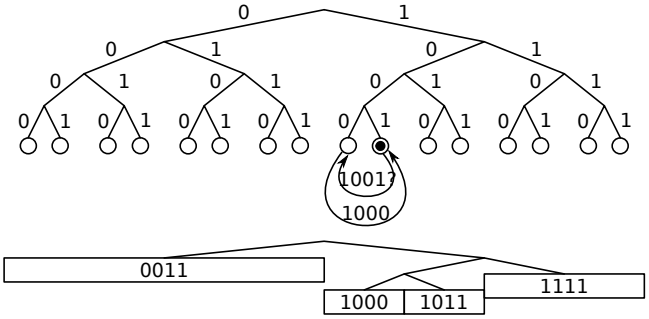

Figure 5.4: Node 1001 adds a link to 1000 and again splits a full bucket. 1001 asks 1000 for 1 link to a node closest to 1001. Since 1000 does not know any closer node, it replies with the link to itself. Node 1001 is now fully connected.

5.3.1 XOR metrics

P2P DHT system Kademlia is based on a knowledge of distance between single nodes. The distance between two nodes is defined as XOR operation of their *nodeID*. We can use a complete binary tree to represent distances of single nodes in a space of node identifiers. Leaves of the tree represent possible *nodeIDs*. Kademlia uses a space of identifiers of a size of 2^{160} identifiers. For simplicity we will show only 2^4 identifiers in this paper.

Each node starts with one bucket, during the time the number of buckets increases to up to 160 - the space of all node identifiers is 160 bits long. In this 160 bits space each two nodes are distant from each other $\langle 2^0;\ 2^{160})$. With the maximum amount of buckets (160) i-th bucket contains up to k links near its node in a range $\langle 2^i;\ 2^{i+1})$, where $i \in \langle 0;\ 160)$. Demonstration on a small space is in the figure 5.5.

When a link to a new node is inserted into routing table and a corresponding bucket is found, there are 3 options:

1. If the bucket contains less than k links the new link is inserted.

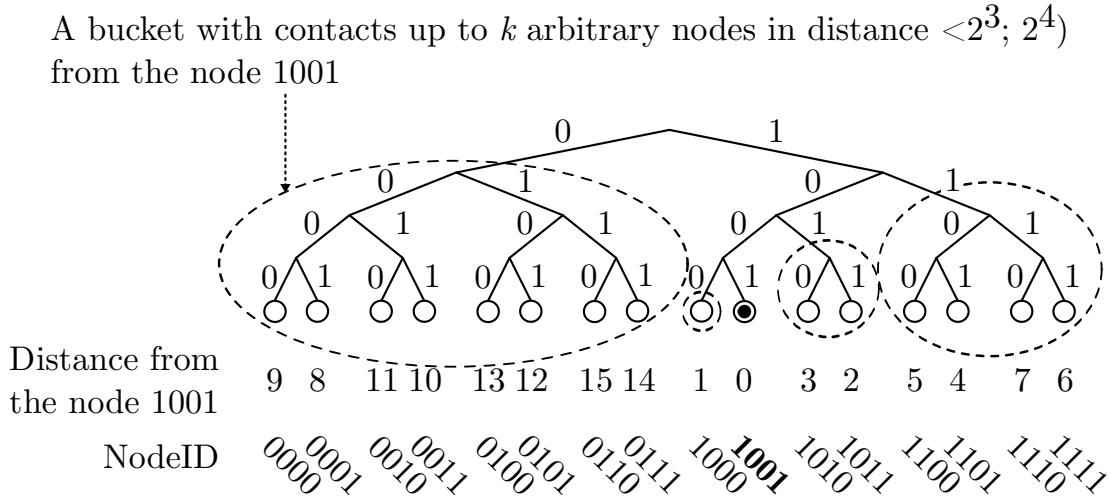

Figure 5.5: Node 1001 keeps links to k nodes from each oval.

2. If the bucket contains k links and it is not a bucket with closest nodes, the node will be ignored.

3. If the bucket contains k links and it is a bucket with closest nodes to its node, the bucket will be decided into two new buckets and the space of identifiers will be split between them.

After initialisation each node has only one bucket that covers the whole space of node identifiers. During discovery of new nodes and further communication the bucket is divided. Initially the bucket covers $\langle 0;\ 2^{160})$, after first divide it splits into two buckets - $\langle 0;\ 2^{159})$ and $\langle 2^{159};\ 2^{160})$. As a node receives any message it updates its routing table. Nodes in buckets are sorted due their last communication with a corresponding node. Buckets are being updated during the network communication - if some bucket hasn't been updated in an hour, it is refreshed. The refresh is done by choosing a pseudorandom *nodeID* of node range that is searched in the network.

5.3.2 Overview of Kademlia implementations

There are several open-source implementations of Kademlia, in Ruby or C++. This chapter analyzes the current state of the art.

Maidsafe-route [56] is developed by Maidsafe.net in C++ and is available under the GPL3 license. Maidsafe-Route is a robust library that implements the DHT algorithm based on Kademlia. It implements various NAT traversal techniques and fulfils all requirements but (F5) in Section 5.2.2. All the communication goes through an API module, but the API module does not allow direct access to IP addresses and ports of single nodes, which is required by Clondike to allow system process migration.

Libcage [57] is a small C++ library. It implements Kademlia protocol with NAT traversal based on the UDP Hole Punching [58]. The main disadvantage is an absence of forced connection to other nodes in a network ((F5) in Section 5.2.2).

CodeMonkeySteve DHT [59] is a Kademlia library implemented in Ruby. It lacks the feature for searching other nodes in a network ((R5) in Section 5.2.2). In the internal implementation it holds all nodes in one array, but according to Kademlia specification it should be kept in single buckets. The library does not allow asynchronous parallel queries.

5.4 Design decision

We have analyzed the architecture of structured P2P systems and their bootstrapping and routing scalability and decided to choose the Kademlia's bootstrapping protocol. Kademlia routes and connects new nodes in $O(\log_2 N)$ steps, but more importantly it requires zero communication messages for a node to disconnect. This is an important behaviour as P2P networks suffer often from sudden node disconnections - a requirement of a proper disconnection routine would cause failures in an unmanaged P2P cluster. The analysis revealed that none of existing Kademlia implementations satisfied all F1-F5 functional requirements specified in section IV B. Therefore, we decided to write our own extended implementation in the Ruby language based on the Kademlia specification.

5.5 Implementation

Implementation of the Kademlia bootstrapping protocol into the Clondike was written in the Ruby language as an extension to existing user space scripts of Clondike. The detailed description is available in [14].

5.5.1 Modifications with respect to the original Kademlia

Clondike implementation does not exactly follow the specification of the Kademlia protocol. We have made following differences in our implementation:

1. We have not implemented remote procedures STORE(key, Val) and FINDVALUE($h(key)$), h is a hash function, since Clondike does not use the Kademlia DHT mechanisms for storing any data. Only procedures PING and FINDNODE were implemented. See [53] for a description of these procedures.

2. By design, Kademlia limits the number of nodes to which a node can connect at a given time to a constant. This is a protection against DOS attacks. But in our experiments, we sometimes need a node to connect to the whole cluster, so we implemented an option to temporary disable this limitation. This will not be used in a production environment.

5.5.2 General usability

The implementation is added into Clondike as an extra module that is general enough to be implemented into any such system. Implementation results of this work are freely available at [14].

5.6 Experimental results

5.6.1 Measurement

The goal of the measurement is to obtain results on a large scale. The new bootstraping process is implemented in Clondike Chameleon, but it runs in a university lab on a real hardware. We have no access to thousands of real hardware machines to run large scale measurements. Instead, we have developed a single-purpose large-scale simulation infrastructure.

We have calculated the number of messages for the former broadcast solution. The total number of messages of the bootstraping process is a key benchmarking metric of each protocol. This theoretical model was compared with a measurement on the newly developed large-scale simulation infrastructure.

The new implemented solution with Kademlia could not be analysed theoretically, because the number of messages depends on many factors, e.g., race conditions in the bucket placement policy, so only measurements on the large-scale simulation infrastructure are provided.

5.6.2 Large-scale simulation infrastructure

In order to obtain results on a large scale, most measurements use virtual image machines instead of measuring on a real hardware. Modern virtualization systems use OS partitioning or para-virtualization, where a user has either no access to the OS kernel or the OS kernel requires further modifications to run in such a system. Clondike contains a patch on the kernel level that would have to be integrated with the hosting kernel of the virtualization system (in case of para-virtualization). This is possible, but this measurement does not require to do an actual OS process migration between individual kernels, since the object of our research is only the bootstrapping process.

This process runs entirely in user space, so we have extracted only user space libraries and created a simulator for bootstrap testing purpose only. We hooked all communication functions on the level of the Ruby framework and all messages going trough these hooks were measured. Thanks to this approach, we were able to run this measurement on a relatively cheap cloud application infrastructure. This simulator was created as a Cloudfoundry [60] application and deployed to IBM Bluemix [10]. The aim was to measure behavior of an at least 4096-node cluster.

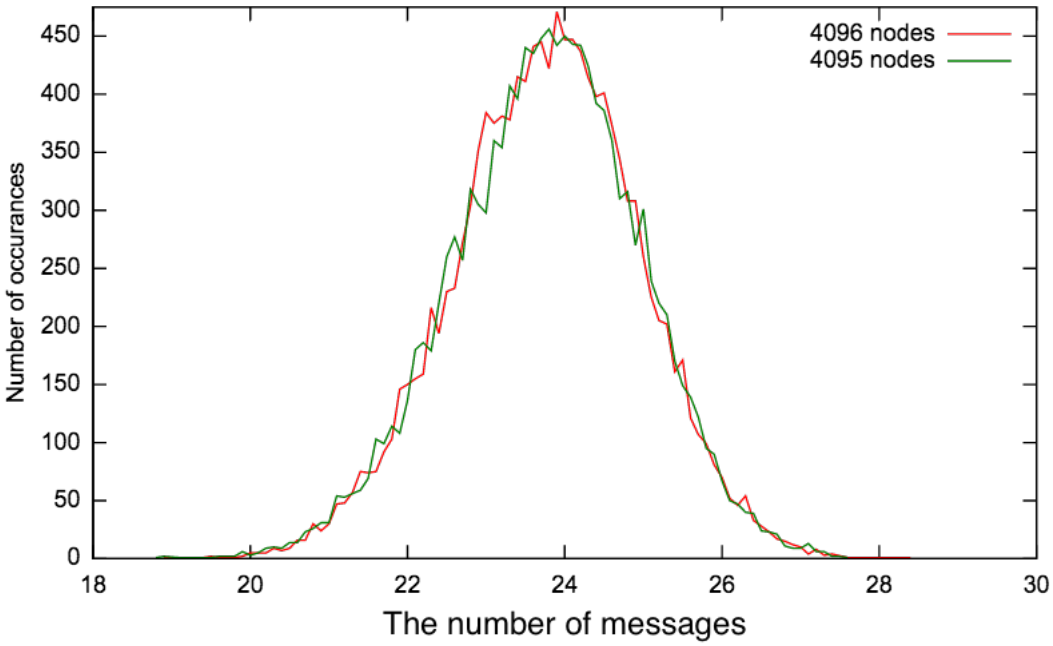

Figure 5.6: Two measurements in detail

The simulator works according to the Kademlia specification. Its function was verified by comparing its simulation results with measurements on real Clondike hardware (see Figure 5.7).

5.6.3 Volatility of results

In the Kademlia protocol, each node has a choice to add a link to a variety of nodes and each of these nodes has links to different nodes. This causes a non-deterministic behavior in the number of messages needed to connect to a cluster since the choice is random. Figure 5.6 shows a distribution of the number of messages during bootstrapping of the 4095th and 4096th node, respectively, if the bootstrap was repeated over 130 000 times. This number of repeats generates a smooth function that copies the normal distribution. Therefore, we have all measurements repeated over 130 000 times and the resulting value is an average of those values.

5.6.4 Replication parameter

In the Kademlia protocol, there is a system-wide replication parameter called k. This is an upper limit of the number of contact information (links) to other nodes that every node stores in its bucket. Most reliable systems in production environment use $k = 20$ [53].

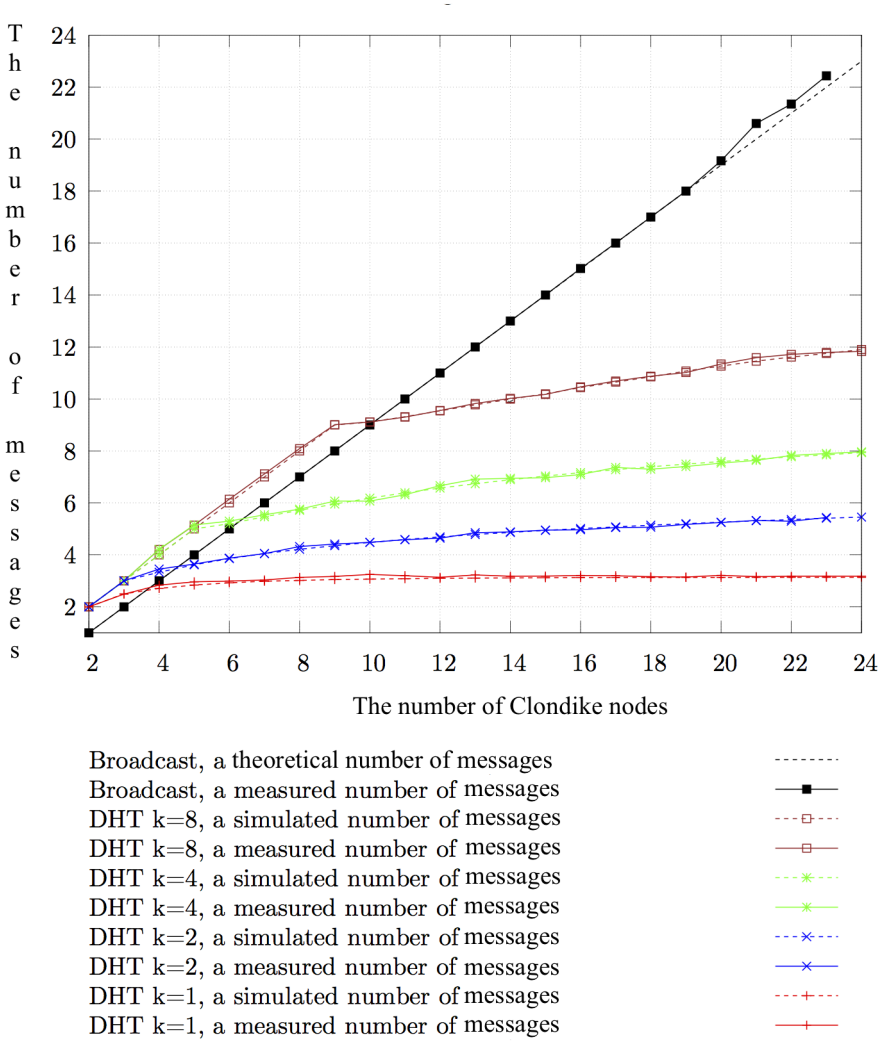

Figure 5.7: Experimental results on real Clondike hardware

However, Clondike does not aim to be rock-solid reliable, it prefers to scale well with a low overhead, like the BitTorrent network. BitTorrent uses $k = 8$ [61]. We have measured different k values on a small number of instances and compared it to the old broadcast solution that scales linearly. Results are in Figure 5.7. Differences with lower k value are quite low so we have decided to stay with the default value $k = 8$ for further measurements.

5.6.5 Measurement results

Figure 5.8 shows that the Kademlia scales logarithmically. We are able to connect the 4096th Clondike node to an existing network using 24 messages in average.

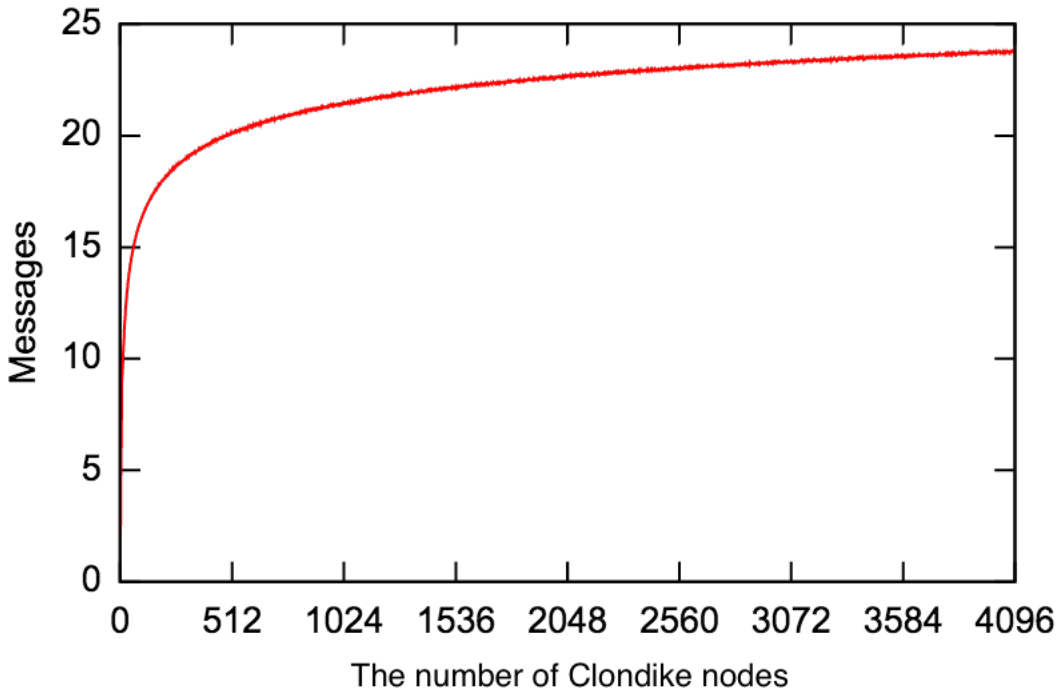

Figure 5.8: Experimental results on a large-scale simulation platform

5.6.6 General impact

The bootstraping process isn't a task specific only to Clondike. Every P2P distributed system needs to solve this issue. Our solution is suitable for any system, where the bootstraping process in a network of up to 4096 nodes must be finished under 25 exchanged messages.

Figure 5.8 shows the average number of messages generated by the Kademlia bootstrapping protocol, the methodology of the measurement was again based on 130 000 times repeated bootstrapping process for all numbers of Clondike nodes ranging from 1 to 4096. These results clearly demonstrated that the Clondike Kademlia protocol scales logarithmically, as was expected. The graph in Figure 5.8 demonstrates that the Clondike Kademlia protocol used 24 messages to bootstrap the 4096th Clondike node to an existing cluster.

5.7 Conclusions

We have analyzed multiple P2P network architectures that provide bootstrapping and inter-node communication. The analysis identified Kademlia to be the best solution for

Clondike since it disconnects nodes without any inter-node communication. We have analyzed existing Kademlia implementations and decided to create our own implementation, since none of them provided all features required for Clondike. Our implementation was evaluated on a large scale of 4096 nodes and the measurement verifies logarithmic scalability of the Kademlia protocol. We needed less than 24 messages (on average) to bootstrap the 4096th node to a cluster. It removes the existing limitations of the Clondike Vanilla project and introduces Clondike Chameleon that allows us to achieve the goal to build a pure P2P scalable computing cluster. Implementation results of this work are freely available at [14] and can be implemented into any other P2P distributed system.

Chapter 6

Unfakeable decentralized system for node reputation evaluation

6.1 Introduction

Fairness issues were in previous Clondike versions (Vanilla, Chameleon) kept aside, as the clusters were still running within one organization or were focused on solving a different problem.

When all nodes belong to a single owner, the motivation of each node is to get the job done. But with opening the cluster to a wider public this scheme changes. The natural behaviour of each node is now to get the most resources from others for the minimal amount of provided own resources. Effective regulations must be introduced.

We introduce a multi-level reputation scoring system that achieves fair usage of resources among all nodes of an inter-organization cluster. The goal is to identify and eliminate nodes that tend to overuse resources of the whole cluster and do not contribute by their computation resources or contribute by false results.

Fair usage policy is an issue in many contemporary P2P file-sharing protocols on a very large scale, such as Bittorrent [5]. But data-sharing or data-exchange systems are in an advantage as data traffic is easily measurable unlike computation resources.

In this chapter we specify requirements for such a reputation scoring system, discuss possible solutions, design an appropriate reputation system, discuss implementation into Clondike cluster and provide an experimental evaluation of the implemented system on a Clondike cluster with abusive players to verify our solution. We introduce this as a new version - Clondike Coypu.

6.1.1 Migration in Clondike

Subjects of migration are standard system processes of Linux operating system. There are two participants in each process migration. Home node is the node where the process was run by a user. Each node has an identity represented by a public key and a private

key. First of all, Clondike load balancer on the home node decides if it is worth to migrate the process (known small processes such as `ls` are skipped). In second step, it chooses an appropriate HOST NODE where the process will emigrate. Host node receives an IMMIGRATION REQUEST and can decide, wherever to accept the process or notify the home node that the migration was rejected. As the host node accepts the immigration request the process is migrated and the home node is notified by EMIGRATE success. From now on, the system process runs on the host node (with all inputs/outputs redirected to the home node). There is no way how to inform the home node how (if) the process finished. This is a major limitation that comes from the design of migrating unmodified system processes. For the purpose of this paper we call a MIGRATION this whole process. A subject of a migration is a TASK. A task represents a single system process. More details can be found at [62].

6.2 Problem statement

Clondike project is released as open source, so anyone can build and run a node with a modified behavior. This scheme is known as `open multi-agent system`. The problem of open multi-agent system is that individual nodes can not be trusted as each node may play its own game. There are following solutions to this problem:

6.2.1 PTM

Pervasive Trust Management Model [63] is based on trust relations between entities. A trust can be gained on a direct or an indirect basis. The overall trust score is calculated on a reference model as the average of all recommendations weighted by the trust degree of the recommender.

6.2.2 An integrated trust and reputation model for open multi-agent systems

FIRE [64] is a complex model that computes the reputation from different sources of trust information direct experience, witness information, role-based rules and third-party references provided by the target agents. It is however build on two assumptions on top of an open multi-agent system agents are willing to share their experiences with others and agents are honest in exchanging information with one other. In Clondike cluster we cant be sure about both of these.

6.2.3 TACS, a Trust Model for P2P Networks

TACS [65] is an bio-inspired algorithm based on a system used in ant colonies where other ants follow once established paths. If an P2P node requests a service it asks the community and receives a path that leads to the optimal node that offers the requested service. This

is useful for heterogeneous P2P networks with many different services (not the case of Clondike).

6.2.4 Problem statement

As discussed below, there exist many solutions of trust management in an open multi-agent system. We define our own reputation scoring system that benefits from the blockchain technology in order to simplify the solution.

6.3 Design

This section discusses the proposed design of the reputation scoring system.

Unlike other models discused in 6.2 we dont base a trust on a relationship between two nodes or a willingness of single nodes to do anything. Clondike is a computation cluster so instead of building a strong interpreted trust based on a feedback of single nodes we prefer to have an information about behaviour of every single node that can be considered as trusted. Each node can use this information and interpret the results independently. We base the trust system on a relation between a node and a system - same as cryptocurrencies. The system itself represents a master authority known from non-distributed systems. But the master authority is distributed and master-less, thanks to the blockchain technology.

6.3.1 Vermins

A Vermin can be found in one or many of following the use cases:

- (UC1) passively reject a migration request (ignore),
- (UC2) actively reject a migration request,
- (UC3) send a wrong task result,
- (UC4) delay or do not provide a task result,
- (UC5) overload the cluster (send more migration requests than accept),
- (UC6) change identity.

Being a Vermin is not strictly binary, there may by situations where to act like a Vermin is the best possible behaviour. Consider given example - a node is overloaded or malfunctioning (for example shutting down), so it is better to actively reject a task to the home node (UC2) than passively reject (UC1) a task in a timeout or fail to deliver (UC3).

6.3.2 Kudos

To achieve a trust between nodes, we propose that key activities of each node will be logged. These log entries will allow any single node to calculate the reputation of a given node. The problem is with (UC6) as defined in 6.3.1. For example a node could constantly switch identities and by each one accuse another node of (UC1). Other nodes could not verify if the accusation is valid - it is much harder to verify something that someone claims that did not happen. For this reason the reputation is build only on positive feedback. If node B does something beneficial for node A node A gives node B Kudos. Kudos is a public positive feedback in the log system system represented by a value. Kudos is only positive - positive actions are easier to verify by other nodes. Kudos records in the log system are verified by other nodes. This prevents (UC6) to generate fake positive reputation for a friendly Vermin node. Verification of Kudos by other nodes is also necessary as each node can emit as much Kudos as it wants, but only verified Kudos are considered in the reputation scoring. Each node can have its own reputation scoring strategy evaluated from the verified Kudos records.

In order to take into consideration multiple types of behaviour we introduce multi-level scoring with different Kudos values. Following Kudos will be defined, sorted from the least to the most valuable:

- (K1) 1 Kudos - immigration request actively rejected,
- (K2) 10 Kudos - immigration request confirmed,
- (K3) 100 Kudos - task result verified.

6.3.3 Log system

Single records in the log system are represented as immutable assets. Each asset has a life-cycle. There can exist multiple assets of various functions in the log system. An asset that is transferred from node A to node B and is verified by majority of nodes is Kudos of node B.

6.3.3.1 Requirements on the log system

- (F1) list assets of any node,
- (F2) create assets,
- (F3) transfer assets to a node,
- (F4) verify a transfer of asset by m-of-n nodes,
- (NF1) immutability of records,
- (NF2) peer-to-peer master-less architecture,

- (NF3) mechanism against double spent assets,
- (NF4) almost database-grade response times.

These requirements overlap with some functions of the blockchain technology discussed in 2.1.3. Blockchain provides a decentralised immutable data store and satisfies (F1-4) and (NF1-3).

6.3.4 Canary tasks

Verification of the whole migration process and especially (UC3) by a third party node is a non-trivial, or maybe even impossible task (depending on a concrete cluster implementation). Therefor we propose to select a task that can be easily verified by a third party node. The task must be reproducible by any node and its input conditions must be easily specified in the log system and remain the same. Ideal candidates are asymmetric cryptographic functions - their result are hard to achieve but easy to verify. Concrete task is cluster-specific and is called CANARY TASK. Canary tasks are emitted by any node in any interval. Before migrating such task to a host node the home node must create a record in the log system where it specifies inputs of a such task. These inputs must be hidden for the host node so the host node does not know it is running a canary task. They can be hidden as a part of other data. The host node receives the same inputs with the migrated process. As the host node actually runs the canary task the home node reveals position of inputs of the canary task in the previous record in the log system. The host node now knows it is running a canary task and submits results into the log system. Other nodes verify the results and if host node success it receives (K3) Kudos. Canary tasks also help newly connected nodes to obtain Kudos.

6.4 Implementation

Theoretical principles were defined in section 6.2.4. Concrete implementation will be discussed in this section.

6.4.1 Blockchain as a storage

Bitcoin blockchain is not suitable to be used as a database system. One of the current issues with Bitcoin is a maximum block size of 1MB that limits the total throughput to 1 transaction per second [66][67]. With the rising number of transactions the latency grows as it takes up to 10 minutes to process each block [68]. Each node of Bitcoin network has to store the whole blockchain that is currently 70 GB at the time of writing [68]. Using Bitcoin as a storage for log system is not suitable as it does not satisfy (NF4) from the requirements specified in section 6.3.3.1.

Bitcoin is an well-established production example of blockchain technology, but requirements specified in section 6.3.3.1 do not actually require a public cryptocurency system.

Other cryptocurrencies (such as Litecoin or Ethereum) based on blockchain could be examinated as well, but the general problem with high latency on blockchain remains.

On the other side there is a number of NoSQL databases such as Apache Cassandra [69] or Apache Hbase [70] that fullfill requirements (F1-F3), (NF2) and (NF4) specified in section 6.3.3.1. In a decent cluster the throughput is hunderts of thousants of transactions per second[68]. The problem is with (NF1) and (NF3) as those are (if needed) solved on the application level.

It is necessary to combine architectures of blockchain and NoSQL databases to fulfill the requirements. A new approach combining these two technologies is introduced in BigchainDB [68].

6.4.1.1 Bigchaindb

BigchainDB [68] is a scalable decentralized master-less database with blockchain characteristics.

BigchainDB is implemented as an extra layer on top of a NoSQL database RethinkDB [71]. It uses an underlying database and adds blockchain-like features as hashed blocks, transactions, voting and immutability of records. The performance is near to NoSQL databases with 1 milion writes per second and a sub-second latency [68]. It is an open source project so we can assume it can not be (easily) compromised.

Single records are represented as JSONs. A verified record is called a block. A block must be verified by a quorum of nodes in order to appear in a database. Each block contains an identifier (id) equal to the block hash in Bitcoin.

Each block contains one or more transactions. A transaction is identified by its id and holds its data in an asset. The data can be any JSON array. Each transaction also contains a link to its previous owner and the current owner.

It fullfills all requirements specified in section 6.3.3.1. Both BigchainDB and RethinkDB are opensource projects so it can be easily integrated into any P2P cluster ecosystem. BigchainDB provides the required log system.

6.4.2 Migration life cycle

Before the actual migration begins, the home node finds the most suitable host node. For the purpose of this paper we use a simple round-robin strategy. Each node can have its own strategy based on data it reads from the log system about other nodes (their Kudos). As the host node is chosen, the home node creates a fist `IMMIGRATION_REQUEST` asset in the log system. This asset contains unique system process id, process name, time-stamp and a public key of the host node. The owner is the home node. By BigchainDB design, this asset once submitted into BigchainDB can not be altered or deleted. As the host node receives an immigration request (along with the public key of the home node) on the Clondike level it requests the log system for a list of assets of the home node (F1) as specified in 6.3.3.1. It verifies if there is an `IMMIGRATION_REQUEST` asset with its own public

key. This verifies that the home node publicly claims its immigration request to the host node. Figure 6.1 shows the flow.

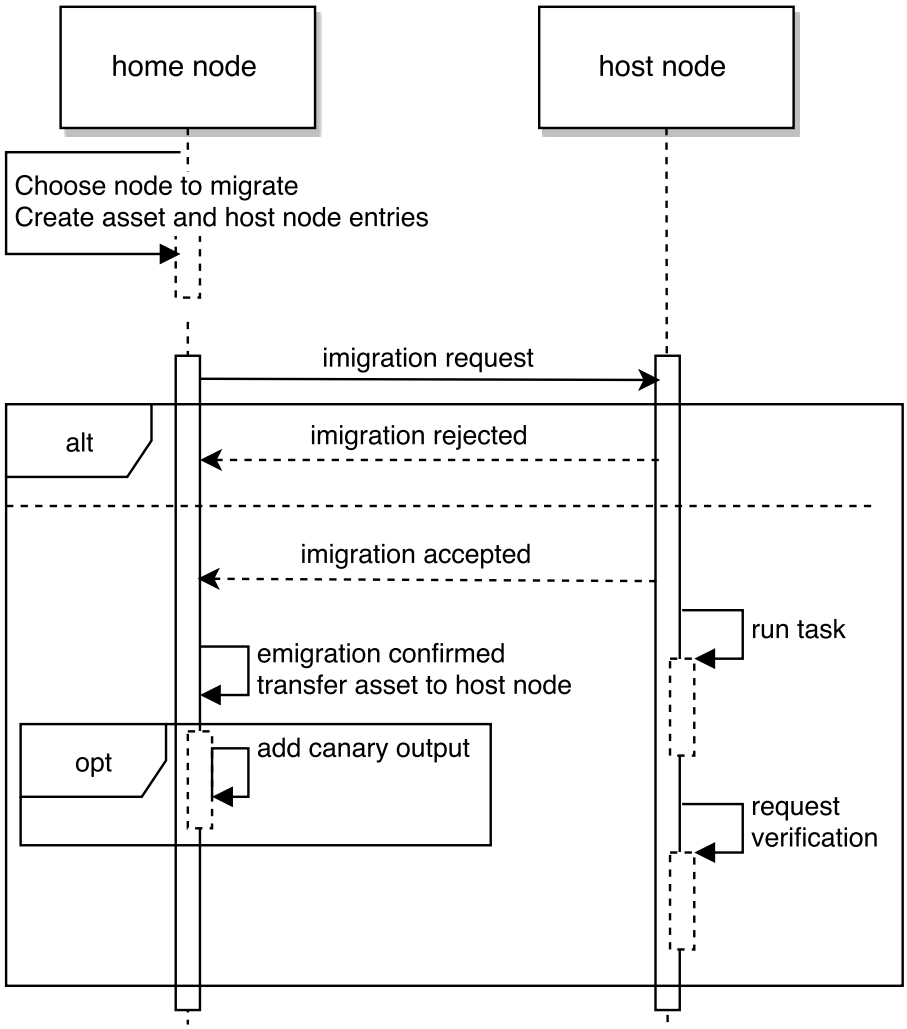

Figure 6.1: Migration life cycle

The next step is a decision of the host node if it accepts the task. It searches through assets associated with the home node from the log system (F1) and evaluates Kudos of the home node. Based on the strategy of the host node it either rejects or accepts the task. The reason for rejection can be its own malfunction or a bad reputation of the home node. Both decisions are (along with existing notification on the Clondike level) submitted into the log system as a new asset of the host node with a reference to the originating `IMMIGRATION_REQUEST` of the home node. New asset is created either as `IMMIGRATION_ACCEPTED` or `IMMIGRATION_REJECTED`.

Now the home node waits until the process is successfully emigrated to the host node. If successful home node creates itself a new asset `IMMIGRATION_CONFIRMED` that references

to the IMMIGRATION_ACCEPTED asset of the host node. There is nothing more the home node can check (the process is no longer running at the home node).

6.4.2.1 Kudos log entry

The host node finishes the task. It notifies the home node (on the Clondike level). The home node should initiate a transfer of IMMIGRATION_CONFIRMED asset to the host node that would undergo verification of other nodes (F4) and be evaluated as Kudos of the host node. Due to actual implementation limitations of BigchainDB [72] this solution is not possible yet. The host node instead creates itself a KUDOS asset containing a reference to the IMMIGRATION_CONFIRMED asset and (F4) the verification is implemented by the Clondike team as a hook in BigchainDB. This hook is triggered only by KUDOS assets. Validation strategies can be different and cluster specific. In case of Clondike, there is a shared file-system among all nodes so for example the nodes can check whenever such binary exists on the host node.

6.4.3 Implemented node strategies

For the purpose of this paper, we have implemented the same simple strategy to all nodes. Nodes verify KUDOS asset by following a chain of referenced assets back to the IMMIGRATION_REQUEST. A host node accepts tasks from a home node if the home node has at least half the Kudos. This is the acceptance border for emigrated tasks from other nodes. Each node emits a canary task in a random interval between 1 and 10 regular tasks.

6.5 Results

In this section we present theoretical and experimental results of designed and implemented multi-level scoring system.

6.5.1 Theoretical verification of results

We describe how is the proposed scoring system applied on concrete Vermin use-cases as defined in section 6.3.1.

(UC1) is a node that passively rejects (ignores) migration requests. Such node will not complete any canary task and therefor it will not receive any Kudos and its reputation stays zero (in case of 100% rejects). Other nodes most likely actively reject migration requests coming from this node.

(UC2) There is small Kudos for active rejection, but other nodes find the active rejection marked in the transaction. If the node will not have any other kinds of Kudos nodes with better reputation reject migration requests coming from this node.

(UC3) If some node accepts a task but provides wrong result it becomes Kudos. There is no way to reveal this. But if the task is a canary task (the node has no way how to

find it out before it accepts) it gets no Kudos for canary tasks. If the node will not have any Kudos for canary tasks nodes with better reputation reject migration requests coming from this node.

(UC4) If a node starts playing dead after accepting a task the situation is the same as with (UC3).

(UC5) If a node accepts and completes some tasks, but requests more task from others has some valuable Kudos and it can still have a good global reputation. However a local decision strategy for accepting a migration request of each node can take into consideration bilateral relations. If there is a significant difference between requests coming from a node and completed task completed by this node from a point of view of a host node, the host node may reject further immigration requests.

(UC6) If a node changes its identity (by generating a new public and private key pair) it starts with a zero reputation. Single Kudos are associated with a specific node identified by its public key.

6.5.2 Experimental verification of results

6.5.2.1 Experimental measurement

Measurement consists of a single compilation of Linux kernel with the same configuration as in [31]. Linux kernel compilation consists of many compilations of C files. Each of these compilations is run as a single system process, hence a stand-alone subject of Clondike migration - task as defined in 6.1.1.

6.5.2.2 Experimental environment

The experimental measurement is run on a virtualised 3 node cluster. The cluster is running on a commodity HW machine. Each node is running Clondike, but strategies of single nodes differ. Nodes 1 and 2 are regular Clondike nodes as described in this paper. Node 3 is a Vermin node with (UC5) behaviour as described in 6.3.1. This behaviour is not an extreme case that would be trivial to identify. For the purpose of this experimental measurement it accepts only each third immigration request, however it correctly replies to the home node with `IMMIGRATION_REJECTED`. Each node has started the experimental measurement task at the same time (3 experimental measurement tasks are running in the cluster in parallel). Each node accepts tasks only from nodes that have at least half the Kudos as itself (this is the acceptance border).

6.5.2.3 Experimental results

Kudos gains of each node during the experimental measurement are shown in Figure 6.2. Figure 6.3 shows number of tasks that each node successfully emigrated to other nodes (were run on a host node) during the same measurement. Each dot represents a single task as defined in 6.1.1.

6.5.2.4 Interpretation of experimental results

At the beginning nodes 1 and 2 accept tasks from node 3 as well - the reason is node 3 received (and accepted) a canary task so its initial Kudos jumps to 100. But since it does not accept all tasks it slowly drops below the acceptance border of both nodes and they start to reject tasks from node 3. However node 3 still accepts every third task (even other nodes do not accept its tasks anymore) so its Kudos still increases, just not as quickly as nodes 1 and 2. Since canary tasks are emitted in a random time periods node 3 gained 2 of them in a quite short time-frame in the middle of the experimental measurement and it returned its Kudos back on top of the acceptance border of nodes 1 and 2 so both nodes started again accepting tasks from node 3. The process is not purely deterministic and each measurement can give slightly different results however the main trend is confirmed - nodes that contribute less resources have less resources from other nodes.

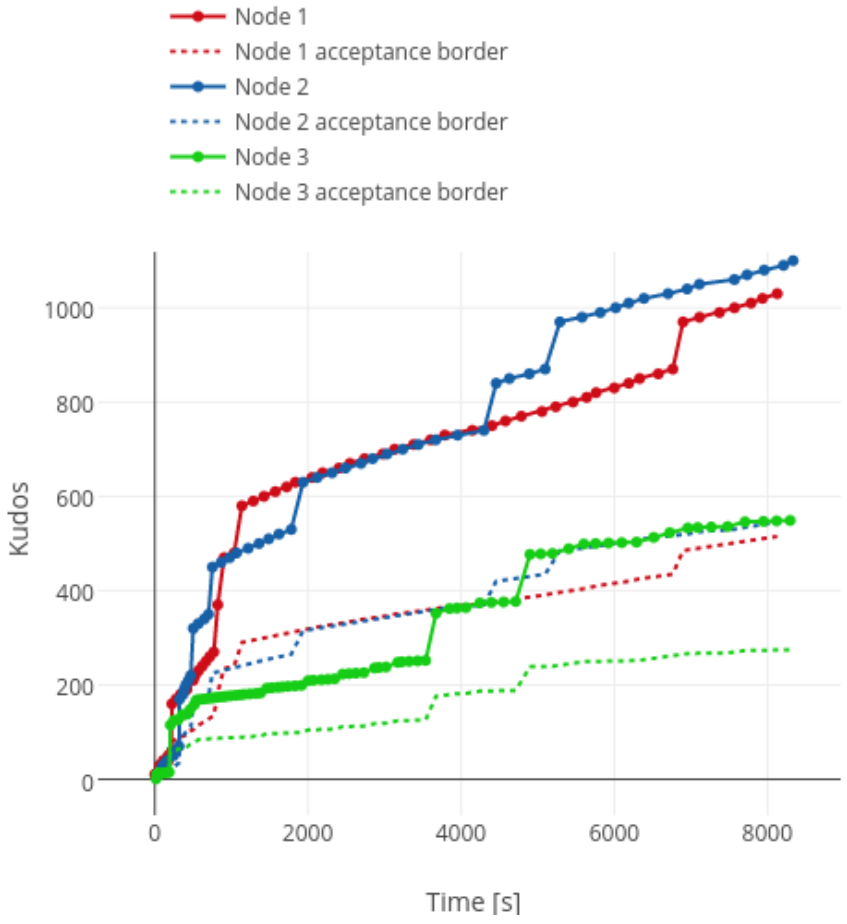

Figure 6.2: Kudos of each node during the measurement task

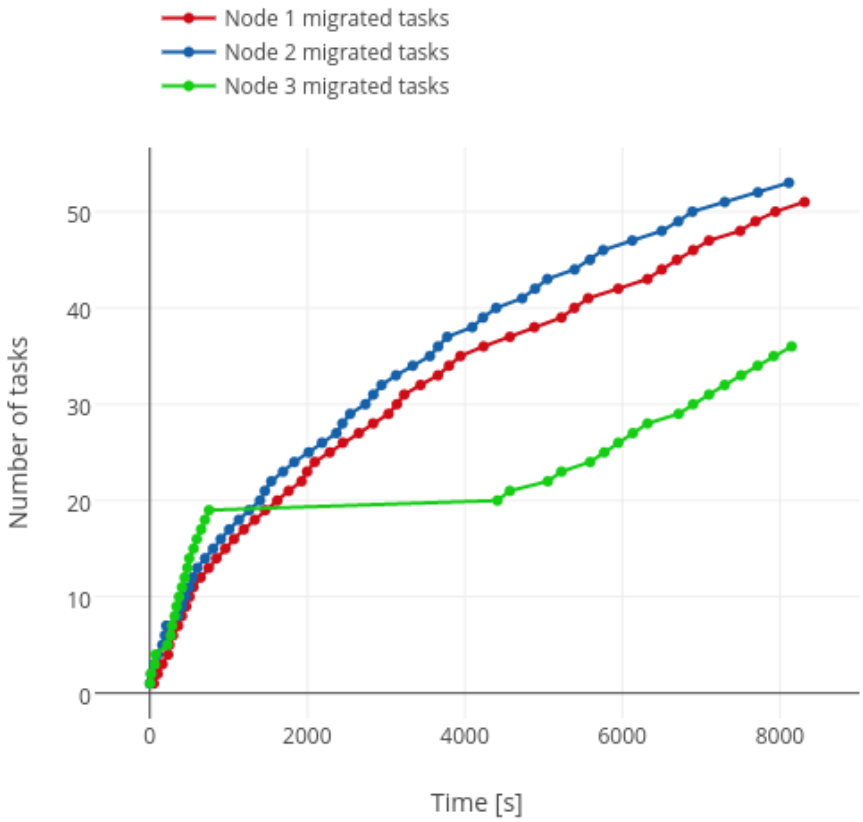

Figure 6.3: Number of task accepted by other nodes during the measurement task

6.6 Conclusions

Distributed multi-level reputation scoring system was designed and implemented and Clondike Coypu was introduced. Its design benefits from the presence of blockchain technology and simplified the open multi-agent system problem. Instead of trusting reputation data that single nodes exchange, each node interprets behaviour data with its own strategy. Behaviour data is verified by a majority of nodes before writing and storing in an immutable distributed blockchain-like log system.

Single abusive use cases were discussed in detail and mechanisms to penalize these use cases were designed and implemented into the multi-level reputation scoring system. Theoretical and experimental verification of these mechanisms were presented. During an experiment on a 3 node cluster both 2 fair nodes successfully penalized an abusive one.

Source codes of our work including the Clondike Coypu release are freely available at [7].

Chapter 7

Conclusions

7.1 Clondike performance scalability on a real use case application

At first we evaluated a scalability of the original version, Clondike Vanilla. We have found a real use case where Clondike can compete with task-specific distributed frameworks. This use case is a distributed source compilation. Concretely it was a distributed compilation of a Linux kernel in C language, but possibilities are almost endless (as there is an almost endless amount of software projects with higher-level source codes).

Source code compilation is a very CPU-intense task, but it still has to exchange a lot of input and output data between home and host nodes. These conditions are far distant from a synthetic CPU-heavy tasks as Mandelbrot computations and show the real strength of a cluster. We have run a series of measurements and found out that Clondike is able to scale with almost the same acceleration as a task-specific framework distcc.

As the next step we evaluated Clondike outside a lab environment in a more realistic network with non-homogeneous nodes. A few Clondike components had to be re-implemented in order to support a such network. Measurement results have shown a limited scalability in this environment and limits of current round-robin-based load balancer.

7.2 Scalable decentralized node discovery and bootstrapping system

One of limiting components in therms of scalability of Clondike Vanilla was node-discovery and inter-node communication. There was implemented a trivial algorithm based on broadcasts over a local network that did not scale.

We have researched this topic in other P2P networks. Clondike is specific as it is a non-dedicated cluster (single nodes can join and leave at any time) and most of the researched

protocols have mandatory routines that need to be executed before a node could properly disconnect.

Best results - logarithmic scalability and zero-disconnection routines - has shown the Kademlia protocol. The main use case for Kademlia is P2P file-sharing and requirements for a such protocol in Clondike were different. Therefore we have designed and implemented our own protocol inspired by Kademlia. Measurements have proved a great scalability of the implemented algorithm. It scales logarithmically and we were able to connect a 4096th Clondike node to an existing network using only 24 messages in average across any network architectures and geographic locations.

We have released this update as a new Clondike version Chameleon.

7.3 Unfakeable decentralized system for node reputation evaluation

The last and the most unknown unresolved problem of Clondike Vanilla was an absence of any controlling mechanisms in terms of fairness among single nodes of the cluster. As Clondike is an open-source project its network suffers from a possible abuse of modified nodes - each node can run a modified version of Clondike. A modified version can (for example) reject all immigration requests. This architecture is called open multi-agent system.

We analyzed existing solutions to this problem but we have chosen a completely different approach. Instead of building a reputation tree from single nodes we have introduced a central log system where single transactions are logged. We have identified use cases that exploit other nodes (called Vermins) and use cases that are beneficial to others. A reputation of single nodes is evaluated from this log system by each node individually. The reputation of single nodes is measured in a newly defined unit Kudos. Kudos is only positive (false-accusations are not an issue). The most Kudos receives a node if it calculates a hidden Canary task.

But as Clondike is strictly a decentralized system the central log system is decentralized as well. This is possible as we utilize an underlying blockchain technology. Each record in the central log system is validated before its subscription by a majority of nodes. Single computations are still kept off-chain and the blockchain acts as a supporting service network. This research result rapidly simplifies a solution of an open multi-agent system problem.

We have released this implementation as a new Clondike version Coypu.

7.4 Summary

We have successfully transformed an existing cluster Clondike closer to the final goal of a global P2P network where single nodes share their computing resources. We have re-

searched an original solution of two existing problems on a field of computing clusters with a completely new approach.

7.5 Contributions of the Dissertation Thesis

All results of our work are implemented in the latest Clondike version and are freely available at a Github repository [7].

7.6 Future Work

The author of the dissertation thesis suggests to explore the following:

- Implementation of blockchain as a service layer for P2P system shows promising results. We propose load-balancing and other unresolved issues should be implemented on blockchain as well in a similar way. This area opens a brand new possibilities as contemporary blockchain technology reaches almost DB-grade response times.
- There are still unresolved issues that must be solved before Clondike can enter any production environment. One of them is security of migrated tasks. Current implementation doesn't inspect the code that is run on a host system what is a security risk. There are three possible approaches - one isolates every single task into a virtual environment (for example an application container), the second does some kind of on-the-fly inspection of the code and the third is based on reputation of single nodes. These approaches can be combined into one strategy.
- Another topic of research is a file-system. Current implementation mounts whole file-system of remote node what is an unacceptable security risk. However research in distributed file-systems has now very promising results and an existing protocol such as IPFS could be used, there is no need to develop a Clondike-specific file-system.

Bibliography

[1] Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system. 2008.

[2] Wikipedia, the free encyclopedia. Bitcoin network data. 2013, [Accessed: 2017-07-23]. Available from: `https://en.wikipedia.org/wiki/Blockchain/media/File:Bitcoin_Block_Data.svg`

[3] Wikipedia, the free encyclopedia. Bitcoin network data. 2013, [Accessed: 2017-07-23]. Available from: `https://en.wikipedia.org/wiki/Blockchain/media/File:Blockchain.svg`

[4] Sotomayor, B.; Montero, R. S.; Llorente, I. M.; et al. Capacity leasing in cloud systems using the opennebula engine. In *Workshop on Cloud Computing and its Applications*, volume 3, 2008.

[5] Cohen, B. The BitTorrent protocol specification, version 11031. 2008.

[6] `https://coinmarketcap.com`, accessed: 2017-07-23.

[7] GitHub - FIT-CVUT/clondike: Projekt Clondike studentu FIT a FEL CVUT v Praze. `https://github.com/FIT-CVUT/clondike`, accessed: 2017-07-23.

[8] Pool, M. distcc, a fast free distributed compiler. 2003.

[9] Cohen, B. Bittorrent dht protocol extension draft.

[10] `www.ibm.com/cloud-computing/bluemix/`, accessed: 2017-07-23.

[11] Kacer, M.; Langr, D.; Tvrdík, P. Clondike: Linux cluster of non-dedicated workstations. In *Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium*, volume 1, IEEE, 2005, pp. 574–581.

[12] `https://sites.google.com/a/fit.cvut.cz/pcg/structure/clondike`, accessed: 2017-07-23.

[13] Štava, M.; Tvrdík, P. Overlapping non-dedicated clusters architecture. In *Computer Engineering and Technology, 2009. ICCET'09. International Conference*, volume 1, IEEE, 2009, pp. 3–10.

[14] GitHub - FIT-CVUT/clondike: Projekt Clondike studentu FIT a FEL CVUT v Praze. `https://github.com/FIT-CVUT/clondike`, accessed: 2017-07-23.

[15] `https://blog.ethereum.org/2016/10/18/faq-upcoming-ethereum-hard-fork/`, accessed: 2017-07-23.

[16] `https://forum.daohub.org/`, accessed: 2017-07-23.

[17] `http://www.coindesk.com/understanding-dao-hack-journalists/`, accessed: 2017-07-23.

[18] Morin, C.; Lottiaux, R.; Vallée, G.; et al. Kerrighed: a single system image cluster operating system for high performance computing. *Euro-Par 2003 Parallel Processing*, 2003: pp. 1291–1294.

[19] `http://www.kerrighed.org/php/clusterslist.php?sort=totalcores&order=desc`, accessed: 2012-01-20.

[20] Morin, C. Xtreemos: A grid operating system making your computer ready for participating in virtual organizations. In *Object and Component-Oriented Real-Time Distributed Computing, 2007. ISORC'07. 10th IEEE International Symposium*, IEEE, 2007, pp. 393–402.

[21] Thain, D.; Tannenbaum, T.; Livny, M. Distributed computing in practice: the Condor experience. *Concurrency and computation: practice and experience*, volume 17, no. 2-4, 2005: pp. 323–356.

[22] `http://linuxpmi.org/`, accessed: 2012-01-20.

[23] `http://www.nasdaq.com/article/bitcoin-price-analysis-bitcoin-hits-gold-parity-again-now-in-usd-cm755817`, accessed: 2017-07-23.

[24] Wilkinson, S.; Boshevski, T.; Brandoff, J.; et al. Storj a peer-to-peer cloud storage network. 2014.

[25] `https://gridcoin.us`, accessed: 2017-07-23.

[26] Namiot, D.; Sneps-Sneppe, M. On micro-services architecture. *International Journal of Open Information Technologies*, volume 2, no. 9, 2014: pp. 24–27.

[27] Merkel, D. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, volume 2014, no. 239, 2014: p. 2.

[28] Buterin, V.; et al. Ethereum white paper. 2013.

[29] He, H.; Fedak, G.; Kacsuk, P.; et al. Extending the EGEE grid with XtremWeb-HEP desktop grids. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference*, IEEE, 2010, pp. 685–690.

[30] EMELYANOV, P. CRIU: Checkpoint/Restore In Userspace, July 2011.

[31] Gattermayer, J.; Tvrdik, P. Different approaches to distributed compilation. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, IEEE, 2012, pp. 1128–1134.

[32] distcc's pump mode: A New Design for Distributed C/C++ Compilation. `https://opensource.googleblog.com/2008/08/distccs-pump-mode-new-design-for.html`, accessed: 2017-07-23.

[33] `http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.32.5.tar.gz`, accessed: 2017-07-23.

[34] `https://github.com/MarstCVUT/Clondike/blob/master/measurements/.config-of-kernel-2.6.32.5-for-test`, accessed: 2017-07-23.

[35] Gunther, N. UNIX Load Average, Part 1: How It Works. Technical report, Technischer Bericht, Performance Dynamics Company, Castro Valley, Kalifornien, USA, 2003.

[36] Google Build Tools Team: Benchmark results for distcc. `http://distcc.googlecode.com/svn/trunk/doc/web/benchmark.html`, accessed: 2012-01-20.

[37] `http://www.samba.org/samba/ftp/stable/samba-3.5.9.tar.gz`, accessed: 2017-07-23.

[38] Gattermayer, J.; Tvrdik, P. Porting clondike to heterogeneous platforms. In *Parallel Distributed and Grid Computing (PDGC), 2012 2nd IEEE International Conference*, IEEE, 2012, pp. 380–384.

[39] PXELINUX. `http://www.syslinux.org/wiki/index.php?title=PXELINUX`, accessed: 2017-07-23.

[40] `http://tools.ietf.org/html/rfc1094`, accessed: 2017-07-23.

[41] `http://lxr.linux.no/#linux+v3.4.4/Documentation/filesystems/tmpfs.txt`, accessed: 2017-07-23.

[42] `http://www.kernel.org/doc/Documentation/filesystems/nfs/nfsroot.txt`, accessed: 2017-07-23.

[43] Esquivel, E. E. B.; Rivero-Angeles, M. E.; Orea-Flores, I. Y. Performance analysis of BitTorrent-like P2P networks for video streaming services at the chunk level. In *Advanced Information Networking and Applications (AINA), 2014 IEEE 28th International Conference*, IEEE, 2014, pp. 806–812.

[44] Gattermayer, J.; Tvrdík, P. Using Bootstraping Principles of Contemporary P2P File-Sharing Protocols in Large-Scale Grid Computing Systems. In *Parallel, Distributed and Network-based Processing (PDP), 2017 25th Euromicro International Conference*, IEEE, 2017, pp. 214–218.

[45] Ranjan, R.; Harwood, A.; Buyya, R. Peer-to-peer-based resource discovery in global grids: a tutorial. *IEEE Communications Surveys & Tutorials*, volume 10, no. 2, 2008.

[46] Clarke, I.; Sandberg, O.; Wiley, B.; et al. Freenet: A distributed anonymous information storage and retrieval system. In *Designing privacy enhancing technologies*, Springer, 2001, pp. 46–66.

[47] Adar, E.; Huberman, B. A. Free riding on Gnutella. *First monday*, volume 5, no. 10, 2000.

[48] Cohen, B. The BitTorrent protocol specification. 2002.

[49] Zhao, B. Y.; Kubiatowicz, J.; Joseph, A. D.; et al. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. 2001.

[50] Rowstron, A.; Druschel, P. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, Springer, 2001, pp. 329–350.

[51] Ratnasamy, S.; Francis, P.; Handley, M.; et al. *A scalable content-addressable network*, volume 31. ACM, 2001.

[52] Stoica, I.; et al. Chord: A scalable P2P lookup service for internet application. In *PROC. ACM SIGCOMM*, volume 1, 2001.

[53] Maymounkov, P.; Mazieres, D. Kademlia: A peer-to-peer information system based on the xor metric. In *International Workshop on Peer-to-Peer Systems*, Springer, 2002, pp. 53–65.

[54] `http://www.vuze.com/`, accessed: 2017-07-23.

[55] Wang, L.; Kangasharju, J. Measuring large-scale distributed systems: case of bittorrent mainline dht. In *Peer-to-Peer Computing (P2P), 2013 IEEE Thirteenth International Conference*, IEEE, 2013, pp. 1–10.

[56] `https://github.com/maidsafe/MaidSafe-DHT`, accessed: 2017-07-23.

[57] `https://github.com/ytakano/libcage`, accessed: 2017-07-23.

[58] Ford, B.; Srisuresh, P.; Kegel, D. Peer-to-Peer Communication Across Network Address Translators. In *USENIX Annual Technical Conference, General Track*, 2005, pp. 179–192.

[59] https://github.com/CodeMonkeySteve/dht, accessed: 2017-07-23.

[60] https://gist.github.com/gubatron/cd9cfa66839e18e49846, accessed: 2017-07-23.

[61] https://www.cloudfoundry.org/, accessed: 2017-07-23.

[62] Novỳ, Z. Implementace škodné do projektu Clondike. 2016.

[63] Almenárez, F.; Marín, A.; Campo, C.; et al. PTM: A pervasive trust management model for dynamic open environments. In *First Workshop on Pervasive Security, Privacy and Trust PSPT*, volume 4, 2004, pp. 1–8.

[64] Huynh, T. D.; Jennings, N. R.; Shadbolt, N. R. An integrated trust and reputation model for open multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, volume 13, no. 2, 2006: pp. 119–154.

[65] Marmol, F. G.; Perez, G. M.; Skarmeta, A. F. G. TACS, a trust model for P2P networks. *Wireless Personal Communications*, volume 51, no. 1, 2009: pp. 153–164.

[66] http://www.coindesk.com/1mb-block-size-today-bitcoin/, accessed: 2017-07-23.

[67] https://en.bitcoin.it/wiki/Scalability, accessed: 2017-07-23.

[68] McConaghy, T.; Marques, R.; Müller, A.; et al. BigchainDB: a scalable blockchain database. *white paper, BigChainDB*, 2016.

[69] http://cassandra.apache.org/, accessed: 2017-07-23.

[70] http://hbase.apache.org/, accessed: 2017-07-23.

[71] Walsh, L.; Akhmechet, V.; Glukhovsky, M. Rethinkdb-rethinking database storage. 2009.

[72] https://github.com/bigchaindb/bigchaindb/issues/729, accessed: 2017-07-23.

Reviewed Publications of the Author Relevant to the Thesis

[1] Gattermayer, J., Tvrdik, P Different approaches to distributed compilation. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW),2012 IEEE 26th International*, IEEE, 2012, pp. 11281134.

The paper has been cited in:

- LIM, Geunsik, et al.: Distributed compilation system for high-speed software build processes. *Big Data and Smart Computing (BIGCOMP), 2014 International Conference*, IEEE, 2014, p. 116-120.

[2] Gattermayer, J.; Tvrdik, P.: Porting clondike to heterogeneous platforms. In *Parallel Distributed and Grid omputing (PDGC), 2012 2nd IEEE International Conference*, IEEE, 2012, pp. 380384.

[3] Gattermayer, J.; Tvrdik, P. Using Bootstraping Principles of Contemporary P2P File-Sharing Protocols in Large-Scale Grid Computing Systems. In *Parallel, Distributedand Network-based Processing (PDP), 2017 25th Euromicro International Conference*, IEEE, 2017, pp. 214218.

[4] Gattermayer, J.; Tvrdik, P. Blockchain-based multi-level scoring system for P2P clusters In *International Conference on Parallel Processing (ICPP)*, IEEE, 2017, in publication process.

Remaining Publications of the Author

[1] Moucha A., Gattermayer, J.: Cluster discovery in phase-shift beamformed ad-hoc and sensor networks. *Wireless Communications and Mobile Computing Conference (IWCMC)*, 2011 7th International.