ENFORCING A GLOBAL RESOURCE QUOTA IN A GRID ENVIRONMENT

KFIR KARMON
ENFORCING A GLOBAL RESOURCE QUOTA
IN A GRID ENVIRONMENT

RESEARCH THESIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS
FOR THE DEGREE OF MASTER OF SCIENCE
IN COMPUTER SCIENCE

KFIR KARMON

SUBMITTED TO THE SENATE OF THE TECHNION — ISRAEL INSTITUTE OF TECHNOLOGY
Tammuz 5769               HAIFA               JULY 2009
ACKNOWLEDGMENT

I am grateful to Prof. Assaf Schuster for his supervision.

THE GENEROUS FINANCIAL HELP OF THE TECHNION IS GRATEFULLY ACKNOWLEDGED
Contents

Abstract 1

1 Introduction 2

2 A Grid-Wide Quota (GWiQ) system 6
  2.1 Grid Wide Resources ............................................. 7
    2.1.1 Resource categories ........................................ 7
  2.2 Grid Wide Quota ................................................... 8
    2.2.1 Sandboxes ................................................... 8
    2.2.2 Resource Coins ............................................... 9
  2.3 Local quota reinforcement ....................................... 11

3 The GWiQ Protocol 13
  3.1 Protocol Overview ............................................... 13
  3.2 Protocol Description ............................................. 15
  3.3 Properties of the GWiQ Protocol ................................ 16

4 Fault Tolerance 25
  4.1 Introduction ..................................................... 25
    4.1.1 Quota Reductions .......................................... 26
    4.1.2 Overlay Deadlocking ........................................ 27
  4.2 GWiQ-P FT .......................................................... 29
    4.2.1 Overlay Deadlocking Elimination ............................ 33
    4.2.2 Quota Reductions Elimination ............................... 34
    4.2.3 Fault Tolerance Augmentation ............................... 43
## Contents

5 Evaluation 50
  5.1 GWiQ-P Properties ............................................. 51
    5.1.1 Scalability .................................................. 51
    5.1.2 Stability ..................................................... 52
    5.1.3 Fast Recovery ................................................ 52
    5.1.4 Fault Tolerance and Balancing Policies ............... 53
    5.1.5 Quota Breach Handling .................................... 57
    5.1.6 Scalability Under Faults ................................. 58

6 Conclusions and Future Work 60

A GWiQ-P Execution Example 63

References 65
List of Figures

3.1.1 Snapshot illustration of a network running GWiQ-P .................. 15
4.1.1 Network setup leading to possible overlay deadlocks ............... 28
4.2.1 Fault tolerance protocol stack ......................................... 30
5.1.1 Scalability evaluation ...................................................... 51
5.1.2 Stability evaluation ......................................................... 53
5.1.3 Fast-Recovery evaluation .................................................. 54
5.1.4 Demand satisfaction in faulty network .................................. 55
5.1.5 Communication costs in faulty network ................................ 56
5.1.6 Quota breach handling ..................................................... 57
5.1.7 Scalability under faults .................................................... 58
A.0.1 GWiQ-P Execution example \( (t = 2, 4) \) .................................. 64
A.0.2 GWiQ-P Execution example \( (t = 7, 8) \) .................................. 65
A.0.3 GWiQ-P Execution example \( (t = 11, 12, 13) \) ......................... 66
List of Tables

3.1 Summary of events handled by GWiQ-P ........................................... 17
4.1 Erroneous scenarios and their possible implication .......................... 26
4.2 FTL’s Properties ............................................................................. 34
Abstract

Mega grids span several continents and may consist of millions of nodes and billions of tasks executing at any point in time. This setup calls for scalable and highly available resource utilization control that adapts itself to dynamic changes in the grid environment as they occur. This work addresses the problem of enforcing upper bounds on the consumption of grid resources. A grid-wide quota enforcement system, called GWiQ-P, is introduced. GWiQ-P is light-weight, and in practice is infinitely scalable, satisfying concurrently any number of resource demands, all within the limits of a global quota assigned to each user. GWiQ-P adapts to dynamic changes in the grid as they occur, improving future performance by means of improved locality. This improved performance does not impair the system’s ability to respond to current requests, tolerate failures, or maintain the allotted quota levels.

The fault tolerant flavor of the GWiQ-P protocol introduces several complementary mechanisms that allow GWiQ-P to be better tuned, in a faulty environment, to performance or strict quota enforcement.

As part of this work a network simulator with the GWiQ-P protocol was implemented. Simulations show the protocol’s properties and strengths on networks containing hundreds of thousands of nodes.
Chapter 1

Introduction

Mega grids, such as EGEE [1] and TeraGrid [2], may span several continents. The vision underlying these huge organizations is the joint management of the collective set of resources at their disposal. Consider a user who, due to a bug in her flow manager, floods the grid with dummy tasks that overburden the resources and deny services to jobs submitted by other grid users. Another user might, out of sheer malice, use the grid to distribute thousands of tasks and launch a distributed denial of service (DDoS) attack. A third user, unaware of the limited number of software licenses available to his organization, overuses the allocated budget by distributing tasks that make use of these licenses.

Clearly, a grid-wide mechanism that enforces the resource quota could solve all these problems and many others as well. The quota system will not allow a user’s tasks to exceed their allocated (or, paid-for) resource budget of CPU hours, RAM gigabytes, storage terabytes, CPUs, blades, disk partitions, database tables, and so forth. The quota system will not allow the user to open too many sockets, send too many messages over the network, send too many bytes, consume too much bandwidth, open too many database connections, use too many IP addresses, etc. The quota system will globally restrict the number of software licenses granted to a user, the total number of authentication operations requested, the inflicted overhead on the encryption servers, and so on.

A single batch system might have some control over resource consumption by the different users. However, the large grids in use today tend to allow the users to choose from a plethora of batch systems such as Condor [3], PBS [4], LSF [5], and Avaki [6],
and may even overlap resource pools. The number of Condor pools in EGEE alone is estimated in the hundreds. The various systems, despite belonging to the same grid organization, and despite potential collaboration (e.g., by means of flocking \cite{7}), do not exchange and do not correlate user information. Furthermore, several state-of-the-art batch systems (e.g., Condor) do not have the necessary mechanisms to enforce an upper bound on the global utilization of the various resource types, even in the scope of a single pool.

The straightforward solution for the quota problem is to have a centralized grid-wide server. Resources are allocated by the server with a lease, to make sure they are returned. A local mechanism on each grid node blocks the resource when the lease expires. However, this solution has several drawbacks. First, the server becomes a single point of failure. Second, the server is not scalable, and becomes congested when the grid scales out. Third, concurrency is reduced, e.g., when thousands of tasks request permission for an available resource at the same time. Fourth, the need to communicate with a remote server, possibly at the other side of the globe, over slow communication lines, may deteriorate performance substantially. Finally, when the resource type is "nonrefundable" (e.g., CPU cycles) and the lease expires due to a crash, it cannot be determined whether the allocated resource was in fact consumed prior to the crash.

In \cite{8}, a hierarchical resource distribution mechanism is presented. The paper only handles the restriction of nonrefundable resources while refundable ones are neglected. Network topology is not considered when transferring quota. Furthermore, fault tolerance is not addressed, thereby causing quota loss when a hosting machine disconnects or crashes. In the work presented in \cite{9} decentralizes PAST's quota system using data replication and multiple servers. This solution is suitable only for PAST networks, and hence cannot be adapted to different network topologies or resource types other than storage. Another network-wide resource restriction mechanism is introduced in \cite{10}. It uses a single management server that dynamically updates the local quotas of distributed sandboxes. The dynamic-policy adjusting server becomes a hotspot and a single point of failure.

This work opts for a highly scalable, highly available, distributed grid-wide quota system. The quota system should operate locally when possible, thus providing immediate permission to use the resource when the utilization level is far from the upper
CHAPTER 1. INTRODUCTION

bound. Still, the quota system must fully enforce the utilization bounds, and must allow budget-level global utilization, even in the presence of extreme erratic behavior, as is commonly experienced with large grids: failures of all types, joins, leaves, crashes and congestion.

In this work, a decentralized grid-wide quota enforcement protocol, called GWiQ-P is presented. GWiQ-P is general in nature and may handle any type of grid resource. GWiQ-P scales implicitly with the size of the grid it is controlling. It is also aware and adaptive to network topology which is utilized to provide better performance. GWiQ-P does not use heavy mechanisms and demanding transactional protocols; it is light-weight, imposing only negligible overhead in terms of computation, communication, and storage. GWiQ-P is highly available, transparently monitoring the utilization level regardless of the dynamic nature of the grid components. The farther the utilization level is from the upper bound, the greater the improvement in performance—because of the foreshortened response time. In fact, GWiQ-P even improves its performance when the system increases its load (in terms of the number of tasks in the grid).

At the heart of our mechanism is the division of a grid-wide quota for a given \((\text{User}, \text{ResourceType})\) pair into discrete quantities called coins. These coins are dispensed throughout the grid, enabling the system to react to dynamic demand changes in a rapid and local manner. The larger the number of coins (relative to the total demand) is, the better the chance to find them locally or in a nearby neighbor. Moreover, the following variant of the locality principle often holds: demand is likely to appear in nodes (or, close to nodes) where it appeared previously. Thus, over time, the protocol moves surplus coins closer to demand hotspots. Alternatively, when demands tend to appear at random locations, the protocol will distribute the surplus coins evenly across the network, over time. In any case, the protocol adapts the surplus coin distribution to the demand pattern.

Our simulated evaluation of GWiQ-P on a grid size of up to one hundred thousand nodes confirms that this scheme scales extremely well. GWiQ-P also shows exceptionally fast behavior in responding to requests in a highly dynamic environment.

An important aspect of this work is the novel fault tolerance flavor of the protocol. Several mechanisms were developed and utilized to support correct behavior and the ability to overcome loss of messages and node crashes. These mechanisms can coexist
and complement each other in the trade-off of strict quota enforcement, on the one hand, and performance on the other.
Chapter 2

A Grid-Wide Quota (GWiQ) system

This chapter provides an overview of the foundational concepts and perception of the GWiQ enforcement task.

In this work, the grid is modeled as a graph, $G(V, E)$ of interconnected nodes, $G$, and edges (or links) $E$. The underlying connected topology is such that a node has a (possibly small) set of neighbors with which it may communicate. A link between two neighboring nodes is not necessarily due to a physically-wired connection but rather may depict the existence of a bi-directional route (which may include nodes that are not part of the grid) by which the neighbors transfer messages. A node’s resource is defined as an exposed service the node is able to provide. Examples of resources are: CPU time, RAM, disk space, network bandwidth, open sockets, etc. A user of the grid may submit numerous jobs, which are binaries executing remotely on the grid’s nodes utilizing their resources.

In chapter 3 a simple variant of a grid wide quota enforcement protocol is displayed. A synchronous, fail free network is assumed. Namely, nodes and links never fail nor do they depart unorderly. Furthermore, the time to transmit a message between two neighboring nodes is finite, bound and known.

In chapter 4, the problem description also includes the possibility of failures both among nodes and links. Furthermore, the network is not assumed to be fully synchronous, rather it is assumed to have a somewhat relaxed asynchronous nature called partial synchrony. This relaxation was described in [11], based on the work in [12],
where several practical models are introduced to overcome the limitations of a fully asynchronous network. Under these asynchronous conditions a sufficiently strong failure detector can be implemented to support the fault tolerant variant of the protocol described in chapter 3.

2.1 Grid Wide Resources

Combining all the nodes and their resources together, the grid can be viewed as a massive supercomputer with vast resources. Given a resource type, the sum of it over all the grid nodes is defined as the grid wide resource. For example, the sum of all the disk-space throughout the grid constitutes the disk-space resource of the grid. Another example of grid wide resources are globally shared logical recourses, for example, floating software licenses or any other external resource that is used by the grid’s nodes.

Definition 2.1 Let $\text{local}\_\text{resource}_i^r(t)$ denote the remaining unallocated amount of resource $r$ at node $i$ at time $t$.

Definition 2.2 The Grid Wide Resource defined for grid $G(V, E)$ and resource $r$ at time $t$ is defined as follows: $\text{GridResource}_r^r(t) = \sum_{i \in V} \text{local}\_\text{resource}_i^r(t)$.

It should be noted that like in any other quota systems, the one suggested here needs not bound the user’s resource consumption to the actual limits due to resource availability. This is trivially done by the resource itself. The quota system springs into action when resource availability is sufficient but the user’s quota is questioned. Therefore, the assumption is made that when a resource is required by a user’s job, either the sandbox will reject the request upfront in case of resource exhaustion, or the quota management solution is consulted on resource availability.

2.1.1 Resource categories

The grid resources can be roughly partitioned into the following two categories: refundable and nonrefundable resources. Nonrefundable resources are resources that, once they have been consumed by user jobs, cannot be consumed again. Therefore,
a user who consumes his or her entire quota for a nonrefundable resource cannot use it further without first receiving an additional quota from the grid managers. (This is done by purchasing further rights to the resource.) Examples of nonrefundable resources are FLOPs, CPU time, and total bytes sent to the network. Naturally, the quota for nonrefundable resources is measured in units of quantity.

Refundable resources are resources that are allocated to user jobs, but once released, are returned to the user’s disposal. Therefore, the quota for refundable resources does not shrink over time. Refundable resources can be quantity-based, such as the total disk storage space granted to a user, or rate-based, such as the maximum combined outbound bandwidth available to all user jobs at a given instant.

This work solves the enforcement of quota restrictions for both refundable and non-refundable resources.

2.2 Grid Wide Quota

A GWiQ is defined as a maximal usage bound a user’s jobs should be globally restricted to, when exploiting a grid wide resource. For example, “John’s jobs should never consume more than 3 Terra-Bytes of disk-space all over the grid at any given time”. A GWiQ can be imposed on every \((User, ResourceType)\) tuple. In the previous example a 3TB GWiQ is set on \((John, DiskSpace)\).

Maintaining any GWiQ for a resource boils down to enforcing subsets of the GWiQ locally at each computing node. This presents two challenges: guaranteeing that the resource utilization of a user-job does not exceed a local quota and ensuring that the sum of local quotas equals the GWiQ at any instant.

2.2.1 Sandboxes

The first challenge can be addressed by well-known sandbox-based solutions for defending the hosting machine’s resources: OS API hooking [13], kernel extensions [14],[15], exploitation of predefined OS hooks [16], and VM-based techniques [17], [10],[18],[19],[20]. Using these techniques, a sandbox can intercept the hosted-job’s resource usage on-the-fly, allowing it to restrict the usage to the local quota. Note that the job does not need to declare its intended usage pattern before it begins running.
The specific architecture and implementation details of such a sandbox are beyond the scope of this work. Therefore, every node in the grid is treated as a sandbox. (The terms 'node' and 'sandbox' will be used interchangeably). Each sandbox maintains a local quota for every resource per user. Any utilization request initiated by a hosted job is intercepted on-the-fly by the sandbox. If the remainder of the resources (the local quota) can satisfy the request, the request is granted and the requested amount is subtracted from the local quota. Otherwise, the request is denied, and the job is suspended until additional quota arrives to its hosting sandbox. When a job frees a refundable resource, it is returned to the local quota. In contrast, consumed nonrefundable resources are never returned.

2.2.2 Resource Coins

The second challenge, of ensuring that the sum of all local quotas equal to the GWiQ, is met by breaking up the GWiQ into equal-sized amounts that are called ”resource-coins” (coins for short), such that the coins’ sum equals the GWiQ. These coins can be distributed freely among the sandboxes. Each coin allows its sandbox to grant a hosted job the use of a grid-wide resource up to the amount that the coin is worth. (Hereafter, the local quotas is expressed in coin units.)

These coins are to be bought by the user from the grid managers. Upon purchase, coins are created for the user for a given resource type by ‘brokers’. Brokers are special nodes connected to the grid.

Thus, for non-refundable resources this scheme ensures that a user ”pays” for a resource before using it, because new coins are introduced to the system only through purchase. This intuitively gives the GWiQ the natural meaning of trading goods. Hence, the total number of coins in the system equals the difference between the total purchased and spent coins.

On the other hand, for refundable resources, the coins act as a security-deposit kept by the sandbox for the duration of the resource usage. Unless new coins are purchased, the number of coins in the system is constant.

To provide a unified solution and explanation for both refundable and non-refundable resources the GWiQ is perceived as the number of purchased coins. Therefore, as long as no new coins are introduced, the GWiQ remains constant. The only factors that
change over time with resource consumption are the amount of "consumed" and "surplus" coins.

Definition 2.3 Let $\text{quota}_{i}^{(u,r)}(t)$ denote the maximal number of coins that user $u$ is allowed to consume of resource $r$, at time $t$, on node $i$.

Definition 2.4 Let $\text{consumed}_{i}^{(u,r)}(t)$ state the worth of coins that user $u$ consumed of resource $r$, till time $t$, on node $i$.

As discussed previously, it is the duty of the sandboxes to uphold the following condition: $\text{consumed}_{i}^{(u,r)}(t) \leq \text{quota}_{i}^{(u,r)}(t)$. Clearly the sandbox will also consider the availability of the resource which is denoted by $\text{local resource}_{i}^{r}(t)$.

Definition 2.5 Let $\text{surplus}_{i}^{(u,r)}(t)$ denote the number of surplus coins available for user $u$ of resource $r$, at time $t$, on node $i$.

Equivalently, $\text{surplus}_{i}^{(u,r)}(t) = \text{quota}_{i}^{(u,r)}(t) - \text{consumed}_{i}^{(u,r)}(t)$

Reiterating, in the special case of refundable resources, a coin gets 'de-consumed' when the resource, worth of the coin, is released by the user’s job. This means that coins jump form $\text{surplus}_{i}^{(u,r)}(t)$ to $\text{consumed}_{i}^{(u,r)}(t)$ on usage, and vice-versa on release. On the other hand, for non-refundable resources, $\text{consumed}_{i}^{(u,r)}(t)$ will increase monotonically on the expense of $\text{surplus}_{i}^{(u,r)}(t)$, which will decrease monotonically.

By using these definitions we encapsulate both refundable and non-refundable resources. This allows us to introduce a unified solution for both.

Definition 2.6 The Grid Wide Quota is defined as the sum of the distributed local quotas. Formally: $\text{GWiQ}^{(u,r)} = \sum_{i \in V} \text{quota}_{i}^{(u,r)}(t)$.

The following is a formal definition of the 'Grid Wide Quota Enforcement' problem:

Definition 2.7 Let $G(V, E)$ be a graph describing the grid, $u$ be a user of the grid, $r$ be a resource that is used by $u$ and exists at least on one node in $V$. Let $\text{GWiQ}^{(u,r)}$ be set to some constant.

A system that claims to solve the Grid Wide Quota Enforcement problem should ensure that, for all time $t$, $\text{GWiQ}^{(u,r)} \geq \sum_{i \in V} \text{consumed}_{i}^{(u,r)}(t)$
CHAPTER 2. A GRID-WIDE QUOTA (GWIQ) SYSTEM

This work claims that solving the ‘Grid Wide Quota Enforcement’ problem can be done by enforcing the local quotas as using the coins scheme as described above. This simple statement can be trivially proved by the following theorem.

**Theorem 2.8** The ‘Grid Wide Quota Enforcement’ problem can be solved by the distributed sandboxes and the resource coins scheme defined above.

**Lemma 2.9** For all $u, r, t, i$ the following is true: $\text{surplus}_{i}^{(u,r)}(t) \geq 0$.

**Proof:** By the definition of surplus we get:

$$\text{surplus}_{i}^{(u,r)}(t) = \text{quota}_{i}^{(u,r)}(t) - \text{consumed}_{i}^{(u,r)}(t)$$

The sandbox, at all time, ensures that: $\text{consumed}_{i}^{(u,r)}(t) \leq \text{quota}_{i}^{(u,r)}(t)$.

**Proof:** Let $G(V, E), u, r$ be the grid’s graph, the user and the resource accordingly, as described in definition 2.7.

\[
\begin{align*}
GWiQ^{(u,r)} &= \sum_{i \in V} \text{quota}_{i}^{(u,r)}(t) \quad (2.1) \\
&= \sum_{i \in V} (\text{consumed}_{i}^{(u,r)}(t) + \text{surplus}_{i}^{(u,r)}(t)) \quad (2.2) \\
&\geq \sum_{i \in V} \text{consumed}_{i}^{(u,r)}(t) \quad (2.3)
\end{align*}
\]

2.3 Local quota reinforcement

Clearly evident from the previous section, though resource consumption is effectively bounded by the suggested mechanism, starvation might occur. This could be due to fragmentation of the quota among the grid nodes. The fragmentation is caused by the breaking the GWiQ into smaller sized coins and dispensing them throughout the grid nodes. Though there might be nodes with surplus coins, other more busy nodes, might exhaust their residue coins. Unused coins in unemployed nodes are a waste of usable quota.
CHAPTER 2. A GRID-WIDE QUOTA (GWIQ) SYSTEM

Definition 2.10 Let \( i \) be a node that identifies a demand of a job of user \( u \) for resource \( r \) at time \( t \). The size of the demand in coins is represented by \( \text{demand}_{i}^{(u,r)}(t) \).

If \( \text{surplus}_{i}^{(u,r)}(t) > \text{demand}_{i}^{(u,r)}(t) \), the entire demand is satisfied and the surplus is reduced. Otherwise, the demand is partially satisfied and a new outstanding demand of size \( \text{demand}_{i}^{(u,r)}(t) - \text{surplus}_{i}^{(u,r)}(t) \) coins is left.

In order to fight starvation of nodes with outstanding demand by nodes with surplus, under GWiQ constraints, a sandbox should pass its unused coins to other sandboxes that have run out of ‘funding’. This constitutes a quota allocation scheme which dynamically adapts to the usage pattern of scattered jobs across the grid, while still adhering to the GWiQ. Possible considerations for determining preferred destinations and quantities of such coin handoffs are discussed extensively in chapter 3.

Consistently it is shown that although refundable and nonrefundable resources have different meanings, they are handled similarly. The only difference is that non-refundable resource coins are eliminated once deposited per resource usage, while refundable resource coins are returned for future use after the resource is freed.

The GWiQ enforcement mechanism suggested in this work system therefore comprises a grid of connected sandboxes, each of which has a demand, a surplus, or neither, at any single moment. The suggested GWiQ protocol aims to route surplus coins to nearby demands, in a decentralized and local manner. These traits make the system extremely scalable and responsive. As will be evident from the protocol properties, the initial location of the coins has no effect on the correctness and over time not even the performance.
Chapter 3

The GWiQ Protocol

This chapter provides a detailed description of the GWiQ protocol. The protocol dictates how surplus coins are passed among sandboxes to satisfy demands. The details of sandbox tasks, such as determining the surplus and demands themselves, are out of the scope of the protocol.

3.1 Protocol Overview

The GWiQ protocol (GWiQ-P) is based on maintaining a spanning forest to achieve local behavior as in [21]. The main idea is to build a small tree around each sandbox with an unsatisfied demand. Each such sandbox acts as the tree’s root, and its demand-less neighboring sandboxes are its descendants. Trees are built in a similar manner to those in Bellman-Ford routing algorithms (BF), using shortest path considerations. However, unlike BF, which maintains a single tree that spans the whole graph (for each root), in GWiQ-P every sandbox participates in a single tree (the one with the closest root), thus forming a forest of distinct, non-overlapping trees. Such a forest overlay will be created for every $\langle User, ResourceType \rangle$ tuple, governing the mobility of coins assigned to that same tuple.

A tree will attempt to transfer any surplus that was encountered on its premises to its root. This can happen either if a new sandbox with some surplus joined the tree, or if a sandbox that already existed in the tree freed a used refundable resource. Basically, any such surplus will be routed along the tree links to the root. However, when a sandbox has passed to its parent enough coins to satisfy the root’s demand,
it will keep any additional coins that it comes by. (Note that the root demand can still be over-satisfied by receiving coins from independent branches.) This scheme encourages on the one hand attraction of coins toward areas with abundant demand, while on the other hand avoids over-concentration of surplus coins in a few number of nodes. Assuming user’s jobs preserve a strong *computation locality* property, such that demands tend to appear close to where others appeared, this makes the coins more available to satisfy future demands in parallel.

Once a demand is satisfied, its tree will be destroyed, allowing every sandbox in the tree to join new trees. Unfortunately, the process of tree destruction is vulnerable to the creation of loops, which may occur in BF algorithms when a node increases its distance to the root \[22\]. (A satisfied demand can be seen as a root that is no longer reachable.) To mitigate this phenomenon, the well-known technique of slowing down information propagation is employed. Specifically, the rate at which new sandboxes can join a demand tree is limited. The event of tree destruction, which is not delayed, will catch up with any sandbox joining the tree, thus eventually eliminating any loops formed.

Concluding the protocol’s behavior, nodes with unsatisfied demands will start forming trees around them. Neighboring nodes, that do not have an unsatisfied demand of their own, join the demand tree with the closest root. Any surplus coins in a tree’s node are sent toward the root and the root’s demand is propagated onwards to neighboring nodes. Nodes that receive a demand request which its root is more distant than their current root, will refrain from joining the tree and from promoting the distant demand, thus avoiding flooding of requests throughout the network. This tendency to confine the request, and prefer to aid within a local environment allow nodes to autonomously satisfy demands among themselves in parallel all over the network. Only in eccentric cases, where either the demand is very large or the close vicinity of a demander lacks sufficient surplus coins, will the tree expand substantially. This comes to show that in the worst case surplus coins will eventually be transferred from one end of the network to the other since any tree can span the entire network if needed.

Figure 3.1.1 visualizes a snapshot of the GWiQ-P’s operation. An example of GWiQ-P in action is supplied in appendix A.
Figure 3.1.1: Example of demand trees and surplus in a grid. Negative numbers represent demands and their growing trees are denoted by the gray circles. Positive numbers represent surplus. A gray ring represents the tree of a satisfied demand in the midst of the tree destruction process.

3.2 Protocol Description

GWiQ-P is described by a set of event-triggered messages that are sent between neighboring nodes. The events are divided into two main categories: environment events and protocol events. The environment events originate either from the networking layer which detects changes in neighbors’ connectivity status or from the resource sandbox when detecting resource demand and relinquishment. GWiQ-P responds to environment events by sending out protocol messages which, in turn, trigger the respective protocol events upon reception.

The protocol messages are also divided into two categories: Overlay-Messages and Quota-Messages. The Overlay-Messages are used to expand and trim trees around the demands. The Quota-Messages are used to convey resource-usage rights.

Table 3.1 lists the events that are handled by GWiQ-P. Algorithm listings 3.2.1-3.2.6 present the full pseudo code of GWiQ-P. It should be noted that an instance of the protocol should be executed in parallel for every \(<User, ResourceType>\) tuple. In the pseudo code listings the \(<User, ResourceType>\) tuple is omitted altogether for clarity.

A node that connects to the grid receives a list of nodes that will serve as its neighbors. The process of selecting these neighbors is not discussed here though any standard mechanism can be used such as peer-to-peer overlays, physical detection using multi-cast messages, etc. Once the networking layer decides on the identity of
the node’s neighbors it fires a LinkUp message for each of them. The same happens on the other side of the link, the node discovers it became another node’s neighbor thus the networking layer fires the respective LinkUp message. In the case of disconnection both sides receive a LinkDown message as if it has been sent by the unreachable neighbor.

After a new link is established, both nodes exchange information regarding their current tree membership by sending a NotifyDemand message. Each node saves a table with the last tree membership information that it received from its neighbors. Upon reception of a NotifyDemand message the node updates the relevant record in the tree membership table. Thereafter, the node calculates if its distance from the new offer’s root is shorter than the one it is currently serving. If the campaigned root is closer, the node changes its membership and sends out similar NotifyDemand messages to its neighbors.

The primal trigger for sending a NotifyDemand message is when a node’s sandbox identifies access to the monitored resource. Then a NeedCoins message is sent locally which triggers the sending of NotifyDemand messages if local quota is insufficient.

Coin passing is triggered by one of three causes: 1) The node joins a new tree and it has surplus coins, 2) the node, which is already part of a tree, frees a local refundable resource and coins are restored and, 3) coins were received from some other neighbor. In all these cases an AcceptCoins message is sent upwards toward the root of the tree.

When a refundable resource is relinquished the sandbox send a DepositCoins message locally, which increase the node’s surplus.

Once a demanding node fully satisfies its demand, a DestroyTree message is sent by the root to all its immediate neighbors. This messages is propagated by all the tree members. When receiving such a message a node detaches from the destroyed tree and chooses the next best tree to connect to out of the tree membership table.

### 3.3 Properties of the GWiQ Protocol

**Correctness** Correctness of this protocol relies on the correctness of a more general algorithm proved in [21]. The following is an intuitive explanation.

Following the cessation of all external events to the protocol (both sandbox and
<table>
<thead>
<tr>
<th>Table 3.1 Summary of events handled by GWiQ-P</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Network Related</strong></td>
</tr>
<tr>
<td><em>LinkUp</em>&lt;sub&gt;i&lt;/sub&gt;(neighbor, weight)</td>
</tr>
<tr>
<td>Node <em>i</em> detected that a new connection was established with node <em>neighbor</em> with measured distance of <em>weight</em></td>
</tr>
<tr>
<td><em>LinkUp</em>&lt;sub&gt;i&lt;/sub&gt;(neighbor)</td>
</tr>
<tr>
<td>Node <em>i</em> detected that the connection with node <em>neighbor</em> has been lost</td>
</tr>
<tr>
<td><strong>Sandbox Related</strong></td>
</tr>
<tr>
<td><em>NeedCoins</em>&lt;sub&gt;i&lt;/sub&gt;(d)</td>
</tr>
<tr>
<td>Node <em>i</em> detected that there is need for <em>d</em> additional coins to satisfy a local resource demand</td>
</tr>
<tr>
<td><em>DepositCoins</em>&lt;sub&gt;i&lt;/sub&gt;(d)</td>
</tr>
<tr>
<td>Node <em>i</em> detected a worth of <em>d</em> coins in resource was released by the hosted job (Note: Relevant only for refundable resources). This event is also used when new quota is purchased from the grid’s broker.</td>
</tr>
<tr>
<td><strong>Overlay Messages</strong></td>
</tr>
<tr>
<td><em>NotifyDemand</em>&lt;sub&gt;i&lt;/sub&gt;(parent, demandID, rootDist, demand)</td>
</tr>
<tr>
<td>Node <em>i</em> is notified by node <em>parent</em> that it knows of demand <em>demandID</em> which is of <em>demand</em> size and <em>rootDist</em> away</td>
</tr>
<tr>
<td><em>DestoryTree</em>&lt;sub&gt;i&lt;/sub&gt;(sendingNode, isTreeDead)</td>
</tr>
<tr>
<td>Node <em>i</em> notified that <em>sendingNode</em> has detached from its tree. If <em>isDeadTree</em> is true, the tree is surely in destruction. Otherwise, connection with <em>sendingNode</em> was lost.</td>
</tr>
<tr>
<td><strong>Quota Messages</strong></td>
</tr>
<tr>
<td><em>AcceptCoins</em>&lt;sub&gt;i&lt;/sub&gt;(sendingNode, c)</td>
</tr>
<tr>
<td>Node <em>i</em> was sent <em>c</em> coins from <em>sendingNode</em></td>
</tr>
</tbody>
</table>
### Algorithm Listing 3.2.1 GWiQ-P Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{surplus}_i$</td>
<td>The surplus that sandbox $i$ possesses</td>
</tr>
<tr>
<td>$\text{parent}_i$</td>
<td>The direct ancestor of sandbox $i$</td>
</tr>
<tr>
<td>$\text{rootDistance}_i$</td>
<td>The sum of all edges’ weights to the root of the tree $i$ is participating in</td>
</tr>
<tr>
<td>$\text{rootDemand}_i$</td>
<td>The demand that is issued by the root of the tree that $i$ is participating in</td>
</tr>
<tr>
<td>$\text{demandID}_i$</td>
<td>A random number identifying of the demand that is issued by the root of the tree that $i$ is participating in</td>
</tr>
<tr>
<td>$\text{treeMembership}_i$</td>
<td>The 4-tuple ${\text{parent}_i, \text{demandID}_i, \text{rootDistance}_i, \text{rootDemand}_i}$</td>
</tr>
<tr>
<td>$\text{passedCoins}_i$</td>
<td>Total number of coins passed through node $i$ in order to satisfy the root’s demand</td>
</tr>
<tr>
<td>$N_i$</td>
<td>Sandbox $i$’s Neighboring Sandboxes</td>
</tr>
<tr>
<td>$e^{i,j}$</td>
<td>The edge connecting sandbox $i$ and sandbox $j$</td>
</tr>
<tr>
<td>$w(e^{i,j})$</td>
<td>The weight of the edge between $i$ and $j$. $(w(e^{i,i}) = 0)$</td>
</tr>
</tbody>
</table>

Assuming a fully synchronous model. This figure can be evaluated on startup.

| $\text{demandID}^j_i$      | $\text{demandID}_j$ as perceived by $i$.                                   |
| $\text{rootDistance}^j_i$  | $\text{rootDistance}_j$ as perceived by $i$.                               |
| $\text{rootDemand}^j_i$    | $\text{rootDemand}_j$ as perceived by $i$.                                 |
| $\text{lastKnownTree}^j_i$ | The 3-tuple $\{\text{demandID}^j_i, \text{rootDistance}^j_i, \text{rootDemand}^j_i\}$ |
Algorithm Listing 3.2.2 GWiQ-P Macros

JoinTree\(_i\)\(\) (parent, demandID, rootDistance, rootDemand)
// Sets the internal variables
  \(\text{treeMembership}_i = \{\text{parent}, \text{demandID}, \text{rootDistance}, \text{rootDemand}\}\);
  passedCoins\(_i\) = 0

JoinBestTree\(_i\)\(\) (isTreeDead)
// Neighbor \(k\) offers the shortest path to a live root
  \(k = \arg\min_v \{\text{rootDistance}\_v^i \mid (\text{demandID}\_v^i \neq \text{demandID}_i) \lor \neg \text{isTreeDead}\}\)
  invoke JoinTree\(_i\)(k, demandID\(_k^i\), rootDistance\(_k^i\) + \(w(e^i,k)\), demand\(_k^i\))
  if surplus\(_i > 0\)
    invoke TransferCoins\(_i\)()
  for all \(v \in N_i\) do
    invoke SendNotifyDemand\(_i\)(v)

DetachFromTree\(_i\)\(\) (isTreeDead)
// Destroy tree and rejoin another
  invoke JoinTree\(_i\)(i, -1, inf, 0)
  for all \(v \in N_i\) do
    invoke DestroyTree\(_v\)(i, isTreeDead)
  if \(\exists k \mid (\text{rootDistance}\_k^i < \infty) \land ((\text{demandID}\_k^i \neq \text{demandID}_i) \lor \neg \text{isTreeDead})\)
    invoke JoinBestTree\(_i\)(isTreeDead)

SendNotifyDemand\(_i\)\(\) (target)
// Delay the NotifyDemand sending by link’s weight
  externInvoke Sleep\(w(e^i,\text{target})\))
  if no change to treeMembership\(_i\)
    invoke NotifyDemand\(_i\)(i, demandID\(_i\), rootDistance\(_i\), rootDemand\(_i\))
Algorithm Listing 3.2.3 GWiQ-P Macros cont.

**TransferCoins\(_i\)()**

// Node \(i\) means to transfer coins via its parent to fulfill \(rootDemand\_i\)
// Note: Variable \(xfer\) is local

\[
\text{if } \text{passedCoins}_i < rootDemand\_i \\
\quad xfer = \text{Min}(\text{surplus}_i, rootDemand\_i - \text{passedCoins}_i) \\
\quad \text{surplus}_i = \text{surplus}_i - xfer \\
\quad \text{passedCoins}_i = \text{passedCoins}_i + xfer \\
\text{invoke } \text{AcceptCoins}_{\text{parent},i}(i, xfer)
\]

**UseCoins\(_i\)(c)**

// Use received coins

\[
\text{externInvoke } \text{Hand over } c \text{ coins to the sandbox} \\
\text{if } rootDemand\_i \leq c \\
\quad \text{surplus}_i = c - rootDemand\_i \\
\quad \text{invoke } \text{DetachFromTree}_i(\text{true}) \\
\text{else} \\
\quad rootDemand\_i = rootDemand\_i - c \\
\text{for all } v \in N_i \text{ do} \\
\quad \text{invoke } \text{SendNotifyDemand}_i(v)
\]

Algorithm Listing 3.2.4 Environment Events - Network Related

**LinkUp\(_i\)(neighbor, weight)**

// This event is called when a new link is created

\[
w(\text{e}^{i,\text{neighbor}}) = \text{weight} \\
\text{invoke } \text{SendNotifyDemand}_i(\text{neighbor})
\]

**LinkDown\(_i\)(neighbor)**

// This event is issued when a link gets disconnected

\[
\text{lastKnownTree}_{i,\text{neighbor}} = \{-1, \infty, 0\} \\
\text{if } \text{parent}_i = \text{neighbor} \\
\quad \text{invoke } \text{DetachFromTree}_i(\text{false})
\]

Algorithm Listing 3.2.5 Environment Events - Sandbox Related

 NeedCoins$_i$(d)
 // Sandbox requested d more coins
 // Note: PseudoRandNum() returns a new pseudo random number
 if surplus$_i$ ≥ d
 surplus$_i$ = surplus$_i$ − d
 externInvoke Hand off d coins to the sandbox
 else
 if parent$_i$ = i
 invoke JoinTree$_i$(i, PseudoRandNum(), 0, rootDemand$_i$ + (d − surplus$_i$))
 else
 invoke JoinTree$_i$(i, PseudoRandNum(), 0, (d − surplus$_i$))
 externInvoke Hand over surplus$_i$ coins to the sandbox
 surplus$_i$ = 0
 for all v ∈ N$_i$ do
 invoke SendNotifyDemand$_i$(v)

 DepositCoins$_i$(d)
 // d coins have been deposited
 // Note: usedCoins is local variable
 if parent$_i$ = i
 invoke UseCoins$_i$(d)
 else
 surplus$_i$ = surplus$_i$ + d
 invoke TransferCoins$_i$( )
Algorithm Listing 3.2.6 Protocol Events

Init$_i()$
// Initialize node $i$
    surplus$_i = 0$
    invoke JoinTree$_i(i, -1, \infty, 0)$
    for all $v \in N_i$ do
        lastKnownTree$_i^v = \{-1, \infty, 0\}$

NotifyDemand$_i$(parent, demandID, rootDistance, demand)
// Node parent has a new demand and it is rootDistance away from its root
    lastKnownTree$_i^{parent} = \{\text{demandID}, \text{rootDistance}, \text{demand}\}$
    if [$(\text{parent}_i = \text{parent}) \land (\text{demandID}_i \neq \text{demandID})] \lor$
            $(\text{rootDistance} + \text{w}(e^{i, parent}) < \text{rootDistance}_i)$
        invoke JoinBestTree$_i$(false)

AcceptCoins$_i$(sender, c)
// $c$ Coins were sent by a child to $i$ in order to fulfill the rootDemand$_i$
    surplus$_i = \text{surplus}_i + c$
    if $\text{parent}_i = i$
        if rootDemand$_i > 0$
            invoke UseCoins$_i(c)$
        else
            invoke TransferCoins$_i()$
    else
        invoke TransferCoins$_i()$

DestroyTree$_i$(sendingNode, isTreeDead)
// This event is called when a neighbor is destroying its tree
    lastKnownTree$_i^{\text{sendingNode}} = \{-1, \infty, 0\}$
    if sendingNode = parent$_i$
        invoke DetachFromTree$_i$(isTreeDead)
network events), it can be said that GWiQ-P is correct if it halts in finite time and upholds the following criterion: if there are any remaining demands, then no surplus exists in the network.

To see informally why GWiQ-P is correct, assume that all external events have stopped. Since no new demands are created, the number of outstanding demands forms a positive non-increasing function. Therefore, there exists a time at which this function is constant, i.e., remaining demands are no longer satisfied. If there are no more demands, then protocol messages will also fade to extinction. (Existing trees are destroyed in finite time, and all coin transfers come to a halt.) Otherwise, each remaining demand becomes a permanent root.

With its roots fixed, the algorithm will create a forest that spans the whole graph in finite time (in the order of the graph’s diameter), due to the correctness of BF. At this point, it follows from the protocol that any remaining surplus will be deterministically routed along tree edges to a root, unless it is blocked by a sandbox along the way.

However, a blocked surplus indicates that the blocking sandbox has already transferred to its parent enough coins to satisfy the demand. These coins either have reached the root, or will reach it in finite time unless they are also blocked. By repeatedly applying the same arguments to sandboxes that are closer and closer to the root, a contradiction is reached to our assumption that roots represent unsatisfied demands at this stage. Therefore, all surplus coins reach roots and (partially) relieve their demands in finite time. Subsequently, the algorithm stops with no surplus at any node.

**Locality** GWiQ-P operates locally in four ways. First, it uses a forest of local trees rather than some global overlay. This way, multiple changes in demand are serviced in parallel without requiring global communication for each operation.

Second, GWiQ-P uses shortest path trees. Hence, nodes containing surplus will aid the demanding nodes that are closest to them. Third, coins tend to be drawn towards the areas of execution. It is reasonable to believe that wherever jobs execute at a given time, future jobs might soon be spawned at the same location. For example, newly submitted jobs are most likely to execute in the user’s cluster unless a group of this user’s jobs flock [7] to a distant cluster. Therefore, such jobs are serviced faster by readily available coins.

Finally, the destruction of a tree takes less time than its construction. As a
result, the tree of a satisfied demand will not expand too far before it is destroyed. This behavior ensures that the instability inflicted by local changes is eased quickly, without causing global effects.

Therefore, it can be conjectured that at most times the coin-passing decisions will be taken on the basis of local information alone, thus allowing GWiQ-P to operate on grids of any size without affecting the efficiency of the solution. This conjecture is validated empirically by means of extensive simulations in chapter 5.

**Distribution** GWiQ-P is a completely distributed protocol, which has no single point of failure. Moreover, its local operation allows it to adapt dynamically to areas of activity. As a portion of the grid increases its activity, the protocol’s operation becomes more local. As a result, hot spots are efficiently avoided. In addition, the initial distribution of the coins when new resources are introduced is of little importance over time.
Chapter 4

Fault Tolerance

4.1 Introduction

In the previous chapter GWiQ-P was introduced to sustain a usage quota of a distributed resource. The protocol achieved this goal in a distributed fashion while it assumed a fully synchronous communication model, in which messages are never lost and are received in the same order they were sent. Furthermore, the aspect of quitting or crashing nodes was not fully considered.

This chapter concentrates the issues that may arise in case the assumptions above do not hold. It also displays the augmentations and changes that need to be applied to GWiQ-P for it to handle these scenarios.

At the base of a distributed protocol is the abstraction of nodes and their links to other nodes. As common in the layered-based network abstraction, there is no knowledge of how this link is created and maintained, whether it is a direct physical connection or one made via a routing messages through routers or other nodes. This increases the variety of possible link related failures and mishaps, from message loss due to a physical link failure or a packet dropped by a mediating router, to message delivery order inconsistencies.

Fortunately, most errors can be categorized as originated from a basic fault which is a link failure. Even the case in which a node crashes or otherwise practices an orderly disconnection form the grid, can be viewed, for most aspects, as if all links connected to that node have been eliminated. Message losses and out of order delivery may also be considered as an abnormal behavior of a link. Hence, the fault tolerant
Table 4.1 Erroneous scenarios and their possible implication

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Possible Implication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node permanent/temporary disconnection</td>
<td>Quota Reductions</td>
</tr>
<tr>
<td>Node crash</td>
<td>Quota Reductions</td>
</tr>
<tr>
<td>Link disconnection while conveying Quota-Messages</td>
<td>Quota Reductions</td>
</tr>
<tr>
<td>Link disconnection while conveying Overlay-Messages</td>
<td>Overlay Deadlocking</td>
</tr>
<tr>
<td>Out of order arrival of Overlay-Messages</td>
<td>Overlay Deadlocking</td>
</tr>
</tbody>
</table>

aspect of GWiQ-P will concentrate mainly, but not solely, on link failures.

As described in 3.2, the protocol messages are divided into two categories: Overlay-Messages and Quota-Messages. The Overlay-Messages group consist of the NotifyDemand messages and DestroyTree messages, which are used to expand and trim trees around the demands. The Quota-Messages group consists only of the AcceptCoins messages which are used to convey resource-usage rights.

This dichotomy projects on the possible implications of a failure on the protocol’s state. Hence, link failures’ implications may also be dichotomized, in this case into: Overlay Deadlocking and Quota Reductions. Namely, a link failure that happens while conveying Overlay-Messages may cause Overlay Deadlocking. Likewise, Quota-Messages affected by a failure will induce Quota Reductions. The special case in which a node crashes or shuts down, aforementioned as equivalent to all its’ links failing, may also induce Quota Reductions independently of the messages on the wire. This special case will be analyzed below.

For a summary of the error prone scenarios and their possible implication please refer to table 4.1.

4.1.1 Quota Reductions

As described above there may be two causes to Quota Reductions: (1) A link failure while conveying Quota-Messages, or (2) A node crash, shutdown or disconnection. These reductions may inflict on the total quota temporally or even permanently depending on the node’s return. Nonetheless, even if temporal they introduce unfairness toward the user.

The following is an example to quota reduction caused by a node crash. The same
claim applies to all cases of node disconnections. At any given time a node may have resource coins in use or kept as surplus. If a node terminates unexpectedly, the coins that reside in that node are lost. This means that the total amount of coins left in the grid is reduced by the surplus (and in the case of a refundable resource, by the number of consumed coins as well).

This issue is amplified in a dynamic networks like peer-to-peer networks. In such networks, nodes join and leave frequently, causing the total quota to deteriorate substantially because of the frequent departures.

Furthermore, the total amount of quota may be permanently reduced as a result of lost Quota-Messages. For example, node $i$ passes an AcceptCoins message to node $j$ granting its root to use 100 coins. If this message never reaches node $j$, either because it is lost during transfer or because node $j$ crashes, the 100 coin grant will be lost for good. Correlating an undesired reduction in the total allowed quota.

### 4.1.2 Overlay Deadlocking

The protocol's implicit assumption is that all messages arrive at their destination correctly and in the same order they were sent. The ordering of the messages is essential to avoid deadlocks in the distributed protocol. These deadlocks can happen in case messages of the Overlay-Messages group get mixed or lost, as will be described in the following example.

Suppose the network has some edge bisecting it into two parts $subnet_1$ and $subnet_2$. Let node $i$ and node $j$ be the two nodes on the bisecting edge, such that node $i \in subnet_1$ and node $j \in subnet_2$. Furthermore, node $j$ has two other neighbors from $subnet_2$, node $k$ and node $l$. Nodes $k$ and $l$ may have other neighbors only from $subnet_2$. node $i$ may also have other nodes only from $subnet_1$. Figure 4.1.1 depicts the network layout.

Suppose node $i$ sees node $j$ as its parent, and node $j$ sees node $k$ as its parent (Not necessarily meaning that node $k$ is the tree’s root). At time $t$, node $j$ receives a DestroyTree message from its root, node $k$, and propagates it to its neighbors (as defined by the protocol). Thereafter, with accordance to the protocol, node $j$ sends a NotifyDemand message to every neighbor $v$ after a delay of $w(e^{j,v})$ (supposedly contradiction to the asynchrony of the system will be discussed in section 4.2). This
message contains the tree information that node $j$ choose to join after leaving the previously collapsing tree. For the sake of the example, assume node $j$ chose to join a tree rooted by node $l$. Let $\text{demandID}_l$ denote node $j$’s new root’s demand.

Being sent before, the $\text{DestroyTree}$ message should reach node $i$ before the $\text{NotifyDemand}$ message. We will show how node $i$ will be deadlocked in case the two messages above reach in the opposite order. Namely, assume that the $\text{NotifyDemand}$ message will arrive to node $i$ before the $\text{DestroyTree}$ message. According to the protocol, when a node receives a $\text{NotifyDemand}$ message from its root with a different $\text{demandID}$ the node reevaluates its options to which tree it should connect. Let’s assume that $i$ re-joins the old-new tree and updates its records (Invoking the $\text{JoinTree()}$ macro). This will cause node $i$ to continue to point to node $j$ as its parent. Thereafter, the old $\text{DestroyTree}$ message arrives from node $i$’s root (node $j$) which will cause node $i$ to detach from the tree pointed by node $j$, although that new tree isn’t collapsing.

We will assume the following assumptions: (1) No new demands are created in the entire network. (2) There are sufficient coins in the entire network to satisfy $\text{demandID}_l$. (3) There are not enough coins resident in $\text{subnet}_2$ to satisfy $\text{demandID}_l$.

After consuming all coins in $\text{subnet}_2$, $\text{demandID}_l$ will indefinitely remain unsatisfied although sufficient coins exist in $\text{subnet}_1$. This is due to node $i$ not being part of the same tree as node $j$ since mistakenly the last received message from node $j$ was the $\text{DestroyTree}$ message instead of the $\text{NotifyDemand}$ message.

According to the protocol, sending out a $\text{NotifyDemand}$ message occurs only if one

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{deadlocking_example}
\caption{Network layout for the deadlocking example. Mixed order messages sent over the link between nodes $i$ and $j$ cause the deadlock.}
\end{figure}
of the following conditions are met: (1) A new link is created. (2) A newer demand was issued. (3) An existing demand was enlarged, or, (4) A previous demand was satisfied causing a change in tree membership. Since, under the assumptions above, none of these conditions apply generally, and specifically to node $j$’s status, no other NotifyDemand message will be sent by it any time in the future. This will cutoff node $i$ and the entire subnet$_1$ from aiding $demandID_l$.

This deadlock, in which node $i$ does not know of node $j$’s needs and node $j$ does not know of node $i$’s ignorance, was due to the incorrect order of arrival and handling of the DestroyTree message and NotifyDemand message by node $i$.

Similarly, if the NotifyDemand message would have not reach node $i$ altogether, the same deadlock would occur. In a similar scenario, a lose of a DestroyTree message can cause node $i$ to remain subordinate to node $j$ indefinitely, albeit its demand was met. This can cause existing demands which their root’s more distant than node $j$’s previous root was, to be left unsatisfied for good.

### 4.2 GWiQ-P FT

The GWiQ-P Protocol will be slightly changed for it to be fault resistant. The augmented protocol, with the fault tolerance support, will be named GWiQ-P FT. Aside from the minor changes to the original protocol a new layer will be introduced that will provide reliable message-passing and neighboring node failure detection. The new Fault Tolerance Layer (FTL for short) will be comprised of well known building blocks from the fault tolerance field. Therefore, it will not be described in great specifics, as it is beyond the scope of this work. The FTL will reside between the GWiQ-P FT layer and networking layer. Any message generated by GWiQ-P FT will be passed down to the FTL which in turn will send it via the networking layer with the appropriate augmentations. Refer to figure 4.2.1 for a layered view of GWiQ-P FT, the FTL and their interaction. Refer to algorithm listing 4.2.7 for a description of the new messages introduced by GWiQ-P FT over the ones in GWiQ-P.

GWiQ-P that was introduced in 3 was assumed to work in a fail free, synchronous environment. One of the major changes in this chapter that the system is no longer assumed to be fully synchronous. An inherit problem of asynchronous systems was displayed in [23]. Consensus cannot be achieved in a fully asynchronous environment.
The root of the problem is that one can not be sure if the response of a process is delayed due to the unbound sending time or to the fact that the process has crashed. This fact sprung several initiatives to circumvent the impossibly proof.

Work in [12], [24], define intermediate models between synchronous and asynchronous, *partial synchrony* for short. These models are a relaxed and more realistic models than the pure asynchronous model. Basically, they do not assume unbounded message-travel-time but rather they assume that such a bound, though not know, exists. They display an even more relaxed version in which these bounds, though
not known beforehand, are established by some other unknown time (called GST for Global Stabilization Time).

In [11], the authors have conceived the idea of the Unreliable Failure Detector. This module, external to the distributed algorithm, provides failure detection services to the algorithm. The weakest failure detector they display there was also proved to be good enough to circumvent the impossibility problem. The imperfection of the failure detector they display is due to the false negatives it produces when wrongfully suspecting a process as dead. Implementation suggestions are quite similar to the one suggested in [12].

Another class of failure detectors is displayed in [25]. The failure detector there simulates a crash in a process that was suspected. Hence, realizing the suspicion. This somewhat harsh policy allows the failure detector to be closer to a perfect one in a sense that when the prediction was wrong the reality is changed to fit it.

In GWiQ-P FT there are several aspects that need to be inspected with regards to the asynchronous nature of the environment and the possible failures. In this work, as part of the FTL, an unreliable failure detector is used with the aggressive policy of simulating a crash on suspicion.

Aside for firing events on link connection and disconnection, the failure detector also supplies estimates to the time bound of sending a message to a neighbor. The GWiQ-P FT protocol uses these estimates for a single purpose. One of the possible ways to eliminate cycles created in a distributed BF algorithm is by delaying some of the messages by the edge’s propagation delay. This means that the GWiQ-P FT protocol needs to acquire these figures from the failure detector. After GST, these bounds are never wrong, by definition. Hence, after GST loops are eliminated correctly. The only concern that may arise is until GST is reached - if messages are not delayed sufficiently then loops may form. This is a false concern since if the delay is underestimated by the FT then it will also suspect the neighbor as disconnected and will fire a disconnection event which will surely break any loop. The only implication will be on the liveness of the system that will delay messages more than necessary. Furthermore, even if a loop were to form, by the GST it would be eliminated.

Another way to circumvent the need for the delays’ bounds was to use a different technique to eliminate loops that does not need these figures. In [21] a different approach was taken in which a three phase tree destruction mechanism was used as
described earlier in [22]. This allows the algorithm to work in a purely asynchronous environment. Aside from this difference, this solution delays the destruction of trees while in GWiQ-P the creation is delayed. Furthermore, the delay in that solution is larger than in GWiQ-P and since this work prefers to target a practical environment in which partial synchrony is enough, the less demanding solution was preferred.

*Overlay Deadlocking* issues are caused due to lost or unordered delivery of *Overlay Messages*. This will easily be handled by adding a *Numbering and Retransmitting* component to the FTL. Each GWiQ-P FT message will be appended with message number to ensure correct reordering at the destination and simple retransmission requests. The need for retransmission is identified by the acceptance of a message with a higher message number than expected. In this case the FTL will request a retransmission and stall the messages that have arrived out of order. After receiving the retransmitted message the FTL will reorder the messages according to their correct order and propagate them to the GWiQ-P FT layer. This scheme can detect any lost message unless it was the last message sent by the neighbor. As described in the previous section, a single missing message, especially if it is the last one sent, can cause overlay deadlocks and quota reductions. To combat this, a heartbeat message will be sent between neighboring nodes. Since heartbeats must continuously be sent the protocol’s message will never be the last one sent. Clearly heartbeats do not need retransmissions in case they are received out of order. Furthermore, heartbeats may be piggybacked on GWiQ-P FT messages.

Another trait of the messages numbering is that duplicate messages, due to retransmissions, can be safely eliminated.

The failure detector (also abbreviated by FD) that may use the already sent heartbeats to detect liveness of a link, may also use other mechanisms in parallel. The exact description of the failure-detector is beyond the scope of this work, though the required properties of the FTL and the failure detector are listed in table 4.2. Simply put, the needed failure detector may wrongfully suspect a healthy link but must eventually suspect a dropped link. Note that the FD should detect if a *link* is up or down rather than if a *node* is. As described above, a disconnected node means that all links are suspected as failed. Concluding that a *node* has failed cannot be accomplished by a single node since it is not likely that it knows its failing neighbor’s entire neighbor-set. Nonetheless, the proposed solution suggests a node needs to know
only the state of its links and not of its neighbors.

In extreme cases a message, heartbeat or a GWiQ-P FT message, may not reach its destination in a reasonable time. After several retransmissions, the FTL will give up and assume the link is down. The LinkDown message event will be fired and GWiQ-P FT will handle it (see infra). In any other case the FD suspects the link has disconnected, it will issue a LinkDown message event.

A link may be suspected wrongfully because of network delays (or any other reason). Nonetheless, the issued LinkDown message event will be handled by GWiQ-P FT without causing an inconsistencies.

Each time a link is restored a transactional handshake occurs between both parties residing on the link to establish an Incarnation Number. This number is appended to every sent message (GWiQ-P FT or heartbeat) and is used to ignore message from past incarnations that were sent on the link in the previous incarnations. Failing to do so may cause deadlock in similar out-of-order scenarios as described above. Only after the connection is (re)established between two nodes, and the incarnation number is agreed, will the LinkUp event be fired for GWiQ-P FT to handle. At this point onwards, all messages from past incarnations will be dropped by the FTL and the basic ordering and safe delivery properties described above will be upheld for the new incarnation of the link.

A node, that its neighbor is negotiating with it on a new incarnation number, immediately assumes that a link failure has previously occurred and simulates a LinkDown message event. This asymmetry could be due to only one of the neighbors perception (justified or otherwise) of a past failure. Since unjustified link failure detection are translated into link resets, they do not harm the correctness of the protocol. This safety precaution eliminates any state inconsistencies between two neighboring nodes.

4.2.1 Overlay Deadlocking Elimination

Resetting the link, by a sequence of LinkDown and LinkUp events, will restart the protocol between two neighbors. This ensures that every valid Overlay Message either reaches its destination in an orderly fashion manner or the link is reset and the protocol is restarted. Hence, Overlay Deadlocking issues are eliminated using the
CHAPTER 4. FAULT TOLERANCE

Table 4.2 FTL’s Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Ensured by</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Eventually every message is received as</td>
<td>Numbering, Heartbeats and Retransmissions</td>
</tr>
<tr>
<td>long as both neighbors are alive</td>
<td></td>
</tr>
<tr>
<td>2 Messages are handled in the same order</td>
<td>Numberings, Heartbeats and Retransmissions</td>
</tr>
<tr>
<td>they were sent</td>
<td></td>
</tr>
<tr>
<td>3 Duplicate messages are ignored</td>
<td>Numberings, Heartbeats and Retransmissions</td>
</tr>
<tr>
<td>4 Messages from past incarnations are</td>
<td>Incarnation Numbers, Incarnation Transactional</td>
</tr>
<tr>
<td>ignored</td>
<td>Persistence.</td>
</tr>
<tr>
<td>5 Eventually, all failures are detected</td>
<td>Failure Detector</td>
</tr>
<tr>
<td>6 A link may be wrongfully suspected as</td>
<td>Failure Detector</td>
</tr>
<tr>
<td>failed</td>
<td></td>
</tr>
<tr>
<td>7 Fault detection may be asymmetrical (A</td>
<td>Failure Detector</td>
</tr>
<tr>
<td>fault might only be detected by one of the</td>
<td></td>
</tr>
<tr>
<td>neighbors, or not at the same time)</td>
<td></td>
</tr>
</tbody>
</table>

FTL layer describe above.

4.2.2 Quota Reductions Elimination

During the regular course of the protocol if an AcceptCoins message is lost the retransmission and numbering mechanism described previously will overcome the fault. The only case in which an undelivered AcceptCoins message can cause a quota reduction is if after the message is sent, the link disconnects and then the message gets lost. Even if the link is reestablished, it will have a different incarnation number thus causing the FTL to drop all expired messages.
Although the 'node crash' scenario was previously analogized to all its link being disconnected it should be noted that with respect to Quota Reduction this is not completely true. For instance, if a node has a surplus of 100 coins and it were to crash or even temporally disconnect. During that outage time the surplus is unreachable hence reduced de-facto from the grid quota. This scenario has nothing to do with lost messages or physical link failures.

In order to combat the Quota Reduction issues the following two solutions are presented. Each has its virtues and vices. Finally, a third, hybrid solution will be suggested.

Transactional Coin Passing

The first suggested solutions is a transaction based solution. An AcceptCoins message will be sent as part of a transaction. When a node that has surplus wishes to pass \( c \) coins to its neighbor it’ll reduce \( c \) from its local surplus and start a transaction of passing these coins to the neighbor. If the transaction is successful, namely, the coins have surely reached their destination, the sending node can forget of their existence. Otherwise, in case the transaction fails, the \( c \) coins are returned to the sending node’s surplus. Similarly, the receiving node will add the accepted coins to its surplus if and only if the transaction is completed successfully.

This solution ensures coins are not mistakenly lost when a link disconnect prior to the acceptance of a sent AcceptCoins message. If a link fails during a transaction the sending node will rollback the transaction and return the coins to the surplus. Similarly, the receiving node would ignore the tentatively accepted coins thus avoiding coin duplication.

The exact description of the transaction mechanism is beyond the scope of this thesis. Theoretically, transactions are not possible in a fully asynchronous environment. Nonetheless, in a partial synchronous environment Weak Atomic Commit can be solved by the E3PC protocol [26] using the fairly weak failure detector which is already incorporated in the FTL.

Transaction endure several messages sent back and forth and the use of persistent storage. This uncovers the the first disadvantage of this method - Performance. Due to the high usage of slow resources this solution suffers from performance issues. Moreover, if the surplus coins are not backed-up on persistent storage then when a
Light-Weight Coin Passing

The second solution introduces a light weight mechanism that solves all the issues that arose in the Transaction Coin Passing solution.

It should be noted that this is the only place in which the grid nodes are differentiated from the Coin Brokers. The brokers are entities that must use persistent storage to save their entire state. Due to their special role of being the primal owners of the created coins they must be secured in a different, more traditional manner. This does not mean that they become bottlenecks or single-points-of-failure since there can be as many brokers as the grid owner wants and they do not have to synchronize between themselves. Furthermore, they have very little interaction with the nodes that are actually executing the jobs as time passes. Hence, the overhead due to their involvement is negligible.

The following is the description of the second solution to link and node failures. It should be applied to all nodes including the brokers. The only difference between the brokers and the regular nodes is, as described previously, that the brokers persist their state and upon node-crash restoration they reset their state to the persisted one.

Each node will keep a special counter per link, denoted by $\text{linkCoins}_{ij}$, which is the counter node $i$ keeps attached to the link connecting to node $j$. Each time the node sends coins via this link - the respective counter will be increased by the number of sent coins. When the node receives coins via that link the same counter is decreased by the number of received coins. Effectively, this counter holds the net number of coins passed the respective link from the node to its neighbor via that link. $\text{linkCoins}_{ij}$ is formally defined by definitions 4.1, 4.2 and 4.3.

**Definition 4.1** Let $\text{sent}_{ij}(t)$ be the number of coins sent from node $i$ to $j$ at time $t$.

**Definition 4.2** Let $\text{received}_{ij}(t)$ be the number of coins accepted at node $i$ from $j$ at time $t$. 
Definition 4.3 Let $\text{linkCoins}_j^i(T)$ be the value of the $\text{linkCoins}_j^i$ variable at time $T$. Namely, $\text{linkCoins}_j^i(T) = \sum_{t=0}^{T} (\text{sent}_j^i(t) - \text{received}_j^i(t))$, such that $t=0$ is the last time the link has been reset.

As proved in lemma 4.5, the sum of the counters on both ends of a link represent the net number of coins that have yet reached their destination. The actual amount of coins that are on the wire is defined formally in definition 4.4. Intuitively, consider a link between nodes $i$ and $j$. Assuming all previous coin-passing messages have reached their destination, the counter at both ends should show the same absolute value with opposite signs. Suppose node $i$ sends 100 coins to node $j$. On their departure, node $i$ will increment the respective counter by 100. At this point the counters’ sum of both ends of the link is equal to 100. Only if the message arrives successfully will node $j$ decrease its counter by 100 and that will cause the counters’ sum to be zero again. It should be noted that it is possible for counters at both sides to be positive, for example, in case both sides send coins to each other. On the other hand, both counters are never negative at the same time, as proved in lemma 4.6.

Definition 4.4 Let $\text{COW}_{i,j}(T)$ be the actual number of coins-on-wire between $i$ and $j$ at time $T$. Namely, the number of coins that have been sent, since the last link reset but have yet to be accepted by time $T$ on the bi-directional link between $i$ and $j$. Formally, $\text{COW}_{i,j}(T) = \sum_{t=0}^{T} (\text{sent}_j^i(t) - \text{received}_i^j(t)) + \sum_{t=0}^{T} (\text{sent}_i^j(t) - \text{received}_j^i(t))$, such that $t=0$ is the last time the link has been reset.

Lemma 4.5 The following is always true:

$$\text{COW}_{i,j}(T) = \text{linkCoins}_j^i(T) + \text{linkCoins}_i^j(T)$$

Proof:

$$\text{COW}_{i,j}(T) = \sum_{t=0}^{T} (\text{sent}_j^i(t) - \text{received}_i^j(t)) + \sum_{t=0}^{T} (\text{sent}_i^j(t) - \text{received}_j^i(t)) \quad (4.1)$$

$$= \sum_{t=0}^{T} (\text{sent}_j^i(t) - \text{received}_i^j(t)) + \sum_{t=0}^{T} (\text{sent}_i^j(t) - \text{received}_j^i(t)) \quad (4.2)$$

$$= \text{linkCoins}_j^i(T) + \text{linkCoins}_i^j(T) \quad (4.3)$$
Lemma 4.6 Assuming the FTL above is used the following term hold: 
$COW_{i,j}(T) \geq 0$

Proof:

$$COW_{i,j}(T) = \sum_{t=0}^{T} (sent_{i}^{t}(t) - received_{i}^{t}(t)) + \sum_{t=0}^{T} (sent_{j}^{t}(t) - received_{j}^{t}(t)) \quad (4.4)$$

$$= \sum_{t=0}^{T} (sent_{i}^{t}(t) + sent_{j}^{t}(t)) - \sum_{t=0}^{T} (received_{i}^{t}(t) - received_{j}^{t}(t)) \quad (4.5)$$

If we assume by contradiction that $COW_{i,j}(T) < 0$, then:

$$\sum_{t=0}^{T} (sent_{i}^{t}(t) + sent_{j}^{t}(t)) - \sum_{t=0}^{T} (received_{i}^{t}(t) + received_{j}^{t}(t)) < 0$$

$\Leftrightarrow$

$$\sum_{t=0}^{T} (sent_{i}^{t}(t) + sent_{j}^{t}(t)) < \sum_{t=0}^{T} (received_{i}^{t}(t) + received_{j}^{t}(t))$$

We reach a contradiction - the total number of received coins on the link, since the last reset, cannot be higher then the number of sent coins on that same period of time (Any messages from previous incarnations of the link are eliminated by the FTL).

When a link fails, what should be done is recreate coins at the sum of the counters on both ends of the failing link. However, this clearly is impossible since the link has failed and the sum cannot be computed. Instead, each node does the following: If the counter is positive it generates that same amount of coins, if it is negative the node generates a fictive demand of that size.

Correctness of these scheme is proved in theorem 4.7. Informally, the fictive demand is generated to globally rectify the quota breach which is due to the generous coin creation on the opposite side of the failed link. The fictive demand will draw coins toward it, as other demand do, and when coins reach it they will be eliminated from the network (even in the case of refundable resources). For example, in case the nodes on both ends of a failing link have another link connecting between them, the fictive demand will be satisfied by passing the equivalent number of coins via the backup link. When the fictive demand is created on one side, the generated coins
on the other will be sent to relief the fictive demand. Thereafter they will disappear and the net number of coins that will remain from the process will be the sum of the counters on both ends before the link failure which, as explained above, is equivalent to the exact number of coins that were lost because of the failing link in the first place. Even if there is no such auxiliary link, the generated coins can either find their way through a lengthier path or can be used by other demands, fictive or regular. In any case the fictive demand will consume coins when it comes across them and eventually extinct the fictive demand and the superfluous coins created when the link failed.

Theorem 4.7 Applying the Light-Weight coin passing and fault handling scheme over the FTL ensures that:

- The GWiQ is never reduced.
- If no more failures occur, any breach of the GWiQ due to the fault handling scheme, is eventually eliminated.

Proof: According to lemma 4.5, $COW_{i,j}(T) = linkCoins^i_j(T) + linkCoins^j_i(T)$. Furthermore, according to lemma 4.6, $COW_{i,j}(T) \geq 0$. Thus, either $linkCoins^i_j(T) \geq 0$ or $linkCoins^j_i(T) \geq 0$ (or both). Without loss of generality, assume $linkCoins^i_j(T) \geq 0$.

According to lemma 4.5, $linkCoins^j_i(T) = COW_{i,j}(T) - linkCoins^i_j(T)$. When a link fails, one of the two cases apply:

1. $0 \leq linkCoins^i_j(T) \leq COW_{i,j}(T)$.
   Therefore, $linkCoins^j_i(T) \geq 0$. According to the algorithm, if the counter’s value is non-negative the node generates the same amount of coins. Hence, in this case node $i$ will generate $linkCoins^i_j(T)$ coins and node $j$ will generate $linkCoins^j_i(T)$ coins. In total $COW_{i,j}(T)$ are generated restoring the GWiQ exactly to its correct value.

2. $COW_{i,j}(T) < linkCoins^j_i(T)$.
   Therefore, $linkCoins^j_i(T) < 0$. According to the algorithm, if the counter’s value is negative the node generates a fictive demand with the same size and non-negative counters are handled as described above. Hence, in this case node
i will generate $\text{linkCoins}_{ij}(T)$ coins which are $\text{linkCoins}_{ij}(T) - \text{COW}_{ij}(T) = -\text{linkCoins}_{ij}(T)$ too many - creating a breach to the GWiQ of $-\text{linkCoins}_{ij}(T)$ coins. Fortunately, this breach will be eliminated when the corresponding fictive demand of size $-\text{linkCoins}_{ij}(T)$ is created in node $j$.

Another interesting figure is the sum of all link endpoint counters of a single non-broker node. During the lifespan of the node, the coins that entered his sandbox have two options: (1) To be sent onwards to some other node as it is part of a tree propagating coins to the demanding root. (2) To be used for its own demand. If $c$ coins were accepted via some link the counter on that link was reduced by $c$. According to the first option, the coins were sent onwards, some (perhaps other) link counter was increased by $c$ when the coins were sent through it. Namely, the sum of all counters on that node remains as before the $c$ coins passed through it. If on the other hand the $c$ coins remain at the node, no other counter is changed and the sum of all counter has been reduced by $c$.

Generally, the sum’s absolute value of a single node’s counters depict the net number of coins that are currently in use (in case this is a refundable resource) or kept in surplus at that node. Note that that number is negative since for incoming coins a counter is reduced. Therefore, the sum of the respective link-counters of that node’s neighbors should be at least of the same absolute value (this time positive). It could be higher if there are more coins currently traveling toward that node. This observation implies that when a node fails its neighbors can act, each, as if the link to that node has failed and no other action is needed to return the lost coins to the grid. This ensures that the quota is never reduced, not even temporarily, when a node departs from the grid. This claim stands for refundable resources. On the other hand, in case of non-refundable resources there are two policies that can be admitted: (1) Apply the policy above as in refundable resources. This lets the user benefit of the doubt - if the coins in the node were consumed or just awaiting in surplus. (2) Do nothing and let the user suffer from the loss of surplus.

Another important observation is that all this is true for a non-broker node. A broker should always persist its link-counters as well as its surplus and after a crash it should restore all counters and surplus and simulate failure on all its links. This
will recreate the coins that were previously sent to its neighbors which caused them to create fictive demands upon the broker’s crash.

It can also be shown that the same argument holds not only for a single node but also in case a larger connected component is detached from the network.

Therefore, this solution mitigated both flaws of the Transactional Coin Passing solution, above. No costly use of transactions and persistent storage and no exposure to quota reductions whatsoever. (Neglecting the broker’s state since they are small number of nodes and totally controlled and monitored by the grid manager).

However, as can be seen in proof 4.7, this solution suffers from a potential quota breaches. During the time span between the fictive demand creation and the fictive demand satisfaction be a non-negligible amount of coins might be generated. These coins can be used by regular user jobs which might cause excessive resource utilization. Consider the following scenario, the network has currently zero surplus coins, all \( Q \) coins are in use. Namely the quota utilization at this state is 100%. Suppose some link has on its one side a counter stating 1000 and \(-1000\) on the other. if that link were to fail at this time, one end of it will generate 1000 coins. If the generating node is part of a demanding tree these will surely be transferred to the root and cause a quota breach since the there will be more than \( Q \) coins in use although \( Q \) was the user’s quota.

Fortunately, as proved in theorem 4.7, this breach will be eliminated eventually once the fictive demand of 1000 will be satisfied and the equivalent number of coins are perished. Unfortunately, until that time, the generously generated coins could be abused by the grid-jobs.

**Hybrid Approach for Quota Reduction Elimination**

At the crux of the light-weight approach is the link saturation. The quota breach as described above is due to links passing a high net volume of coins on one direction before they fail. When such a failure occurs the number of coins generated are as the number of net coins passed on the link from the previous link reset. These coins can be abused by regular user jobs before/instead of satisfying the fictive demand. Reducing the number of generated coins will reduce the potential quota breach. This can be done by *balancing* the counters on both ends of a link before it fails.

The balancing process will start a transaction in which the counters’ absolute
value will both be decreased by the same amount. Basically, during the transaction sending \textit{AcceptCoin} messages will be prohibited. This will assure that by the time the transaction is committed the counter at both ends show the same absolute value. This value will be reduced from the absolute value of both counters causing both to reset.

The balancing process must be done in a transaction to ensure quota is not lost or wrongfully enlarged. If one side were to reset its counter while the other would remain intact, the following failure on that link will cause a non balanced state of generated coins versus fictive demands. If the positive signed counter were to reset while the negative remains this will cause the total quota to be reduced over time when the fictive demand will be satisfied after the link has failed. Similarly, if the negative counter were to be reset leaving the positive counter, the forthcoming failure on that link will generate coins causing an unjustifiable quota enlargement. Hence the necessity of the transaction.

The Hybrid Approach will use the light-weight solution during coin transferring and will further introduce two triggers for link balancing. A link will undergo a balancing process either because the predictable time for the next error is close to expire or if the net number of coins passed has passed a threshold.

The threshold is referred to as the \textit{Balancing Threshold} and the factor by which the predicted time till next failure is multiplied by is referred to as the \textit{Balancing Factor}. Keeping the balancing-threshold on the minimal possible size will force a balancing transaction after every coin passing. This is equivalent to the Transactional Coin Passing solution. On the other hand, setting the balancing-threshold and the balancing-factor to infinity will imitate the Light-Weight Coin Passing solution. Any intermediate values for the two triggers will affect the tradeoff between the costly transactions and the possible instability.

Since transactions are introduced, all nodes must now persist their state as brokers had to do previously. This is due to the fact that transactions change the primal ownership of the coins. For example, assume a broker passes coins to its neighbor and then a transaction happens between the broker and that same neighbor. After that transaction the broker’s link-counters are reset and in case the neighbor crashes the coins there are lost for good. This is why it is imperative that the broker’s neighbor will also persist its state.
CHAPTER 4. FAULT TOLERANCE

This means that the hybrid solution also need all nodes to persist their state and to recover from node-failures as brokers did in the light-weight solution. On the other hand, most coin transfers can still be done with single message passing.

4.2.3 Fault Tolerance Augmentation

This section will display the pseudo code that will be augmented to GWiQ-P to cope with failures according to the hybrid solution. For completeness and brevity the entire GWiQ-P algorithm listings are displayed as in Algorithm Listings 4.2.1-4.2.6 with the fault tolerance augmentations framed. Additional events augmented for the fault tolerance solution are listed in algorithm listing 4.2.7.
## Algorithm Listing 4.2.1 GWiQ-P Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{surplus}_i)</td>
<td>The surplus that sandbox (i) possesses</td>
</tr>
<tr>
<td>(\text{parent}_i)</td>
<td>The direct ancestor of sandbox (i)</td>
</tr>
<tr>
<td>(\text{rootDistance}_i)</td>
<td>The sum of all edges’ weights to the root of the tree (i) is participating in</td>
</tr>
<tr>
<td>(\text{rootDemand}_i)</td>
<td>The demand that is issued by the root of the tree that (i) is participating in</td>
</tr>
<tr>
<td>(\text{demandID}_i)</td>
<td>A random number identifying of the demand that is issued by the root of the tree that (i) is participating in</td>
</tr>
<tr>
<td>(\text{treeMembership}_i)</td>
<td>The 4-tuple ([\text{parent}_i, \text{demandID}_i, \text{rootDistance}_i, \text{rootDemand}_i])</td>
</tr>
<tr>
<td>(\text{passedCoins}_i)</td>
<td>Total number of coins passed through node (i) in order to satisfy the root’s demand</td>
</tr>
<tr>
<td>(N_i)</td>
<td>Sandbox (i)’s Neighboring Sandboxes</td>
</tr>
<tr>
<td>(e_{i,j})</td>
<td>The edge connecting sandbox (i) and sandbox (j)</td>
</tr>
<tr>
<td>(w(e_{i,j}))</td>
<td>The weight of the edge between (i) and (j). ((w(e_{i,i}) = 0)). Assuming a partial synchronous model. This figure can be evaluated by the failure detector.</td>
</tr>
<tr>
<td>(\text{demandID}_{i}^j)</td>
<td>(\text{demandID}_j) as perceived by (i).</td>
</tr>
<tr>
<td>(\text{rootDistance}_{i}^j)</td>
<td>(\text{rootDistance}_j) as perceived by (i)</td>
</tr>
<tr>
<td>(\text{rootDemand}_{i}^j)</td>
<td>(\text{rootDemand}_j) as perceived by (i)</td>
</tr>
<tr>
<td>(\text{lastKnownTree}_j^i)</td>
<td>The 3-tuple ([\text{demandID}_j^i, \text{rootDistance}_j^i, \text{rootDemand}_j^i])</td>
</tr>
<tr>
<td>(\text{linkCoins}_i^j)</td>
<td>The net number of coins passed from node (i) to node (j) as perceived by node (i) (Positive value means more coins were sent than accepted, negative otherwise)</td>
</tr>
<tr>
<td>(\text{inTransaction}_i^j)</td>
<td>A boolean flag determining whether node (i) is currently in a link-counter balancing process with node (j)</td>
</tr>
<tr>
<td>(\text{balanceTimeout}_i^j)</td>
<td>The maximal timeout between two consecutive link-counter balancing transactions on (e_{i,j})</td>
</tr>
<tr>
<td>(\text{balanceThreshold}_i^j)</td>
<td>The maximal value (\text{linkCoins}_i^j) can reach before a balancing transaction is forced</td>
</tr>
</tbody>
</table>
Algorithm Listing 4.2.2 GWiQ-P Macros

JoinTree_i(parent, demandID, rootDistance, rootDemand)
// Sets the internal variables
  treeMembership_i = {parent, demandID, rootDistance, rootDemand};
  passedCoins_i = 0

JoinBestTree_i(isTreeDead)
// Neighbor k offers the shortest path to a live root
  k = arg min \{rootDistance^v_i \mid (demandID^v_i \neq demandID_i) \lor \neg isTreeDead\}
  invoke JoinTree_i(k, demandID^k_i, rootDistance^k_i + w(e^{i,k}), demand^k_i)
  if surplus_i > 0
    invoke TransferCoins_i()
  for all v ∈ N_i do
    invoke SendNotifyDemand_i(v)

DetachFromTree_i(isTreeDead)
// Destroy tree and rejoin another
  invoke JoinTree_i(i, -1, inf, 0)
  for all v ∈ N_i do
    invoke DestroyTree_v(i, isTreeDead)
  if \exists k \mid (rootDistance^k_i < \infty) \land ((\text{demandID}_i^k \neq \text{demandID}_i) \lor \neg \text{isTreeDead})
    invoke JoinBestTree_i(isTreeDead)

SendNotifyDemand_i(target)
// Delay the NotifyDemand sending by link’s weight
  externInvoke Sleep(w(e^{i,target}))
  if no change to treeMembership_i
    invoke NotifyDemand_target(i, demandID_i, rootDistance_i, rootDemand_i)
Algorithm Listing 4.2.3 GWiQ-P Macros cont.

TransferCoins$_i()$

// Node $i$ means to transfer coins via its parent to fulfill $rootDemand_i$
// Note: Variable $xfer$ is local
\[
\text{if } \neg \text{inTransaction}_{i}^\text{parent} \\
\text{if } \text{passedCoins}_i < rootDemand_i \\
\quad xfer = \text{Min}(\text{surplus}_i, rootDemand_i - \text{passedCoins}_i) \\
\quad \text{surplus}_i = \text{surplus}_i - xfer \\
\quad \text{passedCoins}_i = \text{passedCoins}_i + xfer \\
\quad \text{linkCoins}_{i}^\text{parent}_i = \text{linkCoins}_{i}^\text{parent}_i + xfer \\
\quad \text{invoke } \text{AcceptCoins}_{\text{parent}_i}(i, xfer)
\]

UseCoins$_i(c)$

// Use received coins
\[
\text{externInvoke } \text{Hand over } c \text{ coins to the sandbox} \\
\text{if } rootDemand_i \leq c \\
\quad \text{surplus}_i = c - rootDemand_i \\
\quad \text{invoke } \text{DetachFromTree}_i(\text{true}) \\
\text{else} \\
\quad rootDemand_i = rootDemand_i - c \\
\quad \text{for all } v \in N_i \text{ do} \\
\quad \quad \text{invoke } \text{SendNotifyDemand}_i(v)
\]

Algorithm Listing 4.2.4 Environment Events - Network Related

LinkUp$_i$(neighbor, weight)

// This event is called when a new link is created
\[
w(e_i, \text{neighbor}) = \text{weight} \\
\quad \text{invoke } \text{SendNotifyDemand}_i(\text{neighbor})
\]

LinkDown$_i$(neighbor)

// This event is issued when a link gets disconnected
\[
\text{lastKnownTree}_{i, \text{neighbor}} = \{-1, \infty, 0\} \\
\text{if } \text{parent}_i = \text{neighbor} \\
\quad \text{invoke } \text{DetachFromTree}_i(\text{false}) \\
\text{if } \text{linkCoins}_{i}^\text{neighbor} < 0 \\
\quad \text{invoke } \text{NeedCoins}_i(|\text{linkCoins}_{i}^\text{neighbor}|) \\
\text{else if } \text{linkCoins}_{i}^\text{neighbor} > 0 \\
\quad \text{invoke } \text{DepositCoins}_i(\text{linkCoins}_{i}^\text{neighbor}) \\
\quad \text{linkCoins}_{i}^\text{neighbor} = 0
\]
Algorithm Listing 4.2.5 Environment Events - Sandbox Related

NeedCoins\(_i(d)\)
// Sandbox requested \(d\) more coins
// Note: \(PseudoRandNum()\) returns a new pseudo random number
if \(surplus_i \geq d\)
    \(surplus_i = surplus_i - d\)
    externInvoke  Hand off \(d\) coins to the sandbox
else
    if \(parent_i = i\)
        invoke  JoinTree\(_i(i, PseudoRandNum(), 0, rootDemand_i + (d - surplus_i))\)
    else
        invoke  JoinTree\(_i(i, PseudoRandNum(), 0, (d - surplus_i))\)
    externInvoke  Hand over \(surplus_i\) coins to the sandbox
    \(surplus_i = 0\)
for all \(v \in N_i\) do
    invoke  SendNotifyDemand\(_i(v)\)

DepositCoins\(_i(d)\)
// \(d\) coins have been deposited
// Note: \(usedCoins\) is local variable
if \(parent_i = i\)
    invoke  UseCoins\(_i(d)\)
else
    \(surplus_i = surplus_i + d\)
    invoke  TransferCoins\(_i()\)
Algorithm Listing 4.2.6 Protocol Events

\textbf{Init}_i() \\
// Initialize node i \\
\text{surplus}_i = 0 \\
\text{if state exists on persistent storage} \\
\text{surplus}_i = \text{stored_val} \\
\textbf{invoke} \ \text{JoinTree}_i(i, -1, \infty, 0) \\
\textbf{for all} \ v \in N_i \ \textbf{do} \\
\text{\quad lastKnownTree}_v = \{-1, \infty, 0\} \\
\text{\quad linkCoins}_{\text{neighbor}} = 0 \\
\text{\quad if state exists on persistent storage} \\
\text{\quad \quad linkCoins}_{\text{neighbor}} = \text{stored_val} \\
\textbf{invoke} \ \text{BalancingStart}_v(i) \ \text{every} \ \text{balanceTimeout}_v \\
\textbf{NotifyDemand}_i(\text{parent}, \text{demandID}, \text{rootDistance}, \text{demand}) \\
// Node \text{parent} has a new demand and it is rootDistance away from its root \\
\text{lastKnownTree}_{\text{parent}} = \{\text{demandID}, \text{rootDistance}, \text{demand}\} \\
\textbf{if} \ [(\text{parent}_i = \text{parent}) \ \land \ (\text{demandID}_i \neq \text{demandID})] \ \lor \ \\
\text{\quad \quad (rootDistance + w(e_{\text{parent}}) < rootDistance_i)} \\
\textbf{invoke} \ \text{JoinBestTree}_i(\text{false}) \\

\textbf{AcceptCoins}_i(\text{sender}, c) \\
// c Coins were sent by a child to i in order to fulfill the rootDemand_i \\
\text{\quad linkCoins}_{\text{sender}} = c \\
\text{\quad surplus}_i = surplus_i + c \\
\text{\quad if parent}_i = i \\
\text{\quad \quad if rootDemand}_i > 0 \\
\text{\quad \quad \quad \textbf{invoke} \ \text{UseCoins}_i(c)} \\
\text{\quad else} \\
\text{\quad \quad \textbf{invoke} \ \text{TransferCoins}_i()} \\
\text{\quad \quad if \ \mid \text{linkCoins}_{\text{sender}} \mid > \text{balanceThreshold}_{\text{sender}} \\
\text{\quad \quad \quad \textbf{invoke} \ \text{BalancingStart}_{\text{sender}}(i)} \\

\textbf{DestroyTree}_i(\text{sendingNode}, \text{isTreeDead}) \\
// This event is called when a neighbor is destroying its tree \\
\text{lastKnownTree}_{\text{sendingNode}} = \{-1, \infty, 0\} \\
\text{\quad if sendingNode = parent}_i \\
\text{\quad \quad \textbf{invoke} \ \text{DetachFromTree}_i(\text{isTreeDead})}
Algorithm Listing 4.2.7 Fault Tolerance Additional Events

BalancingStart\(_i\)(neighbor)

// Node \(i\) was requested by \(\text{neighbor}\) to begin balancing transaction
// Symmetry breaker in case neighbors start a transaction simultaneously
  if \(\text{inTransaction}_{\text{neighbor}} \rightarrow (i > \text{neighbor})\)
    \(\text{inTransaction}_{\text{neighbor}} = \text{true}\)
    invoke BalancingSuggest\(_{\text{neighbor}}\)(\(i, \text{linkCoins}_{\text{neighbor}}\))

BalancingSuggest\(_i\)(neighbor, proposal)

// Node \(\text{neighbor}\) proposed \(i\) that both reduce the link-counter by \(\text{proposal}\)
  \text{linkCoins}_{\text{neighbor}} = \text{linkCoins}_{\text{neighbor}} - \text{proposal}
  invoke BalancingAccept\(_{\text{neighbor}}\)(\(i, \text{proposal}\))

BalancingAccept\(_i\)(neighbor, proposal)

// Node \(i\) receives \(\text{neighbor}\)’s acceptance to reduce the link-counter by \(\text{proposal}\)
  \text{linkCoins}_{\text{neighbor}} = \text{linkCoins}_{\text{neighbor}} - \text{proposal}
  \text{inTransaction}_{\text{neighbor}} = \text{false}
  invoke BalancingEnd\(_{\text{neighbor}}\)(\(i\))
  if parent\(_i\) = \(\text{neighbor}\)
    invoke TransferCoins\(_i\)()

BalancingEnd\(_i\)(neighbor)

// Node \(i\) was requested by \(\text{neighbor}\) to commit the transaction.
// Finally, compensate suppressed coin passing during the transaction
  \text{inTransaction}_{\text{neighbor}} = \text{false}
  if parent\(_i\) = \(\text{neighbor}\)
    invoke TransferCoins\(_i\)()
Chapter 5

Evaluation

GWiQ-P is intended for large-scale grid systems under high load. Therefore, GWiQ-P was simulated on large graphs, focusing on dynamic scenarios. The network topology, on which the simulations were run, were generated using the BRITE [27] topology generator. The generated topologies reflect the main properties of large scale grids: low latency LANs interconnected by links with relatively higher latency. The simulations were focused on refundable resources since they have a more dynamic nature. Over time, non-refundable resource should behave as refundable since they are expected to be repurchased.

The simulations were conducted according to the following scheme. Given a graph of \( n \) nodes, we fix the percentage of demanding nodes, the overall demand, and the GWiQ. The simulation is conducted by randomly changing the identity of the demanding nodes at a constant rate, while keeping the overall demand constant. The change rate - specified by the percentage of the overall demand - is relocated every average edge delay. Subsequently, we measure the average resource utilization in a sliding window in steady state. Furthermore, initial demands are set to 100 coins. In the fault-tolerance scenarios a fixed percentage of the grid’s links are dropped every average edge delay. A dropped link is restored after a period equivalent to its latency has passed.

The default values that were used are: 10K nodes with a BriteAS network topology, 1% of all nodes have demands at any given time, 1% of the overall demand is relocated every average edge delay and the GWiQ is equal to the overall demand. In fault-tolerance scenarios the following defaults were additionally used: 1% of all
links fail every average edge delay, a link balancing threshold was set to 100 coins and the balancing factor was set to 1 (see section 4.2.2 for more details). In any case a simulation description below lacks details regarding one of these properties, the defaults above were used.

5.1 GWiQ-P Properties

5.1.1 Scalability

Figure 5.1.1: Overall demand satisfaction vs. network size. Comparing GWiQ-P with a centralized protocol on mesh and Internet-like (BriteAS) topologies.

Figure 5.1.1 presents the percentage of the satisfied user demand as a function of the network’s size. Both GWiQ-P and a simplistic centralized protocol are evaluated. The centralized solution randomizes, on start up, a location for a central node that does the resource book-keeping. All resource demand requests are sent from the requesting node to the central node. A minimal spanning tree is used for routing.
requests back and forth. Requests are granted as long as the total usage will remain
below the allocated quota. Resources are simulated to be leased for a predetermined
fixed period of time, hence, upon release no communication with the central node is
needed. The lease time is set to have the same characteristics as in the simulations of
GWiQ-P. Furthermore, no bandwidth restrictions are enforced on the routing tree or
the central node. Effectively, the only restriction posed during the central algorithm
simulations is derived from the network topology - the networks’ diameter.

As can be seen from the figure, GWiQ-P achieves the same demand satisfaction
regardless to the grid’s size. This points to the protocol’s scalability due to its locality
property. Locality allows GWiQ-P to be satisfy demands using small neighborhoods
around a demand, making it indifferent to the increase of the network size. On the
other hand, the centralized protocol suffers substantially from the grid size increase.
The centralized protocol had poor results despite no bandwidth restrictions were
applied. The increase of the graph diameter, causing higher average latency from a
random node to the central node, was enough to cause the dramatic results.

5.1.2 Stability

Figure 5.1.2 displays the impact the demand change rate has on the overall satisfac-
tion. Naturally, the overall demand satisfaction is reduced as the demand becomes
more erratic. Nonetheless, the drop in the overall demand seems to be with linear
correlation to the demand change rate (Axises are log-scaled). The figure also shows
how mitigating this problem by supplying excess GWiQ can help, to some extent.
The $\frac{Q}{D}$ Ratio axis signifies the ratio between the supplied GWiQ and the overall de-
mand. For example, if this ratio is set to 4, there is four time more quota than the
overall demand in the grid.

5.1.3 Fast Recovery

Figure 5.1.3, depicts GWiQ-P in a transient behavior scenario. A system of 10K
nodes is simulated until it converges (achieving 100% of the possible utilization).
Subsequently, 10% of the demands are relocated (resulting in an immediate 10% drop
in utilization) and the utilization is measured over time. As depicted in figure 5.1.3,
the system re-passes 99% satisfaction after less than 25ms in all but one case, which
CHAPTER 5. EVALUATION

Figure 5.1.2: Overall demand satisfaction with respect to the rate the overall demand relocates and the $Q_D$ Ratio which is the ratio between the excess GWiQ (Q) and the total overall demand (D).

is substantially less than the graph’s diameter. As described above, the network is modeled as several fast sub networks connected by slower links. The average latency endured by an internal network link is ~2.48ms, while inter-connecting links impose ~23ms delays. According to the protocol, for coins to pass from one node to another at least three times the latency between them must pass. Once for delaying the demand notification, once for passing the demand notification and at last once for sending the coins. Therefore, it is evident that the recovery process, which took only ~25ms, was handled entirely in a small vicinity within the demands’ local networks.

5.1.4 Fault Tolerance and Balancing Policies

Previous figures dealt with simulations that incorporated no faults at all. Figure 5.1.4 displays the satisfied demand percentage in networks that may endure link failures.
Figure 5.1.3: Transient behavior of satisfied demands with time (avg. edge delay = 2.5ms). At t=100, 10% of the demands are relocated. System Size: 10K. Demanding nodes: 10%. Different GWiQ-Demand ratios are evaluated.

Every average edge delay, 1% of all links are dropped for the duration of their latency. Each bar represents an edge balancing policy that is made of two properties: (1) Maximal value the special fault tolerance counter may reach before a balancing transaction is forced (`Balancing Factor`) and (2) The factor that determines the maximal time between two consecutive link balancing transactions (`Balancing Factor`). Setting the balancing threshold to zero forces every coin transfer to be followed by a balancing transaction. While, setting both properties to infinity will allow no balancing throughout the simulation. The plane cutting through the bars depicts the satisfied demand in a scenario with similar properties except that it endures no faults at all.

Figure 5.1.4 shows that when setting GWiQ-P to force a balancing transaction after every coin pass causes a mild reduction in the demand satisfaction with respect to the no-faults scenario. This is evident from the left most bar-column which lays below the cutting plane. The slight reduction is due both to lower connectivity due to disconnection of links and to the blockage of links undergoing balancing transactions after every coin passing.
Figure 5.1.4: Satisfied demand while grid endures ongoing link failures. Several link balancing policies are evaluated. Note that setting the Balancing Factor to zero is effectively forcing each coin transfer to be conducted within a transaction. Setting both properties to infinity will allow no balancing at all. The plane cutting through the bars depicts the satisfied demand in a similar scenario that endures no faults at all.

On the other hand, when balancing is not that aggressive, demand satisfaction is higher compared to the no-faults scenario. The explanation to this counterintuitive observation is that the excess coins created by the light-weight fault-tolerance solution cause a potential GWiQ breach which is exploited by the user’s job. Coins are created for every failing link that its fault tolerance counter is non-zero. Although a fictive demand is generated as well, that will eventually eliminate coins causing the breach, the user’s jobs can take advantage of these coins in the mean time to increase the overall satisfaction. These coin creations also help coins appear, out of thin air, in areas of the network that currently suffer from deficiency. This also helps demands gain coins faster.

Figure 5.1.5 displays the average communication costs. The communication costs
Figure 5.1.5: Average communication costs while grid endures ongoing link failures. Several link balancing policies are evaluated. Note that setting the Balancing Factor to zero is effectively forcing each coin transfer to be conducted within a transaction. Setting both properties to infinity will allow no balancing at all. The plane cutting through the bars depicts the number of messages an average node sends during a period of an average edge delay in a similar scenario that endures no faults at all.

are defined by the number of messages sent on average by a node during a period of an average edge delay. Clearly, all scenarios produce more messages than in a non-fault system (as depicted by the plane cutting the bars). The additional message are due to: 1) The messages sent on link reconnection, 2) the additional handling of fictive demand trees and 3) the messages generated by the transactions. The communication penalty in case no balancing is used at all is higher than in the case transaction based coin passing is used. This is mainly due to the numerous fictive demand trees that are created. Nonetheless, the average number of messages is very low in absolute terms
as it is bound by 3 messages per 2.5ms (average edge delay) for a node.

5.1.5 Quota Breach Handling

![Graph showing transient behavior of potential GWiQ breach](image)

Figure 5.1.6: Transient behavior of the potential GWiQ breach while handling faults (avg. edge delay = 2.5ms). From t=100 till 1100 links are dropped at a fixed rate of 1% of the edges every average edge delay. Throughout the simulation 1% of the demands are relocated every average edge delay. System Size: 10K. Demanding nodes: 10%. $Q/D$ Ratio = 0.5. Different balancing policies are evaluated. The curves are ordered in the legend in the same order they appear.

Figure 5.1.6 shows how the potential GWiQ breach is affected over time by various fault handling policies. As discussed previously, the coins that are recreated after a link failure can be exploited by user’s jobs. The creation of part of these coins is justified since some of the coins were lost during the link failure. On the other hand, the majority of the coins are actually created as a side effect, hence they unjustifiably increase the GWiQ. This amount of coins will be eliminated by the fictive demand eventually. Until that happens, the user’s jobs might take advantage of the excess
coins for their needs. Figure 5.1.6 displays the potential quota breach and the time it takes for the fictive demand to eliminate it. The potential breach is calculated as the total number of coins that exist in the grid divided by the original GWiQ. Using several link balancing policies it can be seen that this potential breach can be significantly reduced by increasing the rate in which link-balancing transactions are employed up to total elimination in case transactional coin passing is used. Furthermore, it can be seen that the system recovers very fast from the breach, created by a set of link failures. This is due to the numerous trees that spread around the fictive demands which clean up the duplicate coins effectively.

5.1.6 Scalability Under Faults

![Nodes' Tree-Sizes Distribution](image)

Figure 5.1.7: Distribution of the nodes’ tree sizes. Each node is connected to exactly one tree at any given time (singleton trees are the default state). As the overall demand increases proportionally to the network size, the GWiQ does also - conserving the $Q/D$ Ratio in all simulations.
Every node is part of exactly one tree at any given time. As can be seen in figure 5.1.7, the distribution of tree sizes each node is connected to, remains similar no matter what the network size is. This shows that GWiQ-P is indifferent to the network’s size due to its strong locality characteristic. This figure reenforces and extends the conclusion that GWiQ-P is infinitely scalable, as previously seen in figure 5.1.1, also in scenarios with ongoing link failures.
Chapter 6

Conclusions and Future Work

This work tackles one of the acute problems of grid systems deployed today - the difficulty in enforcing a global quota. GWiQ-P, efficiently enforces a grid-wide quota while maintaining several valuable properties: it can support a wide range of grid resource types; it is fully distributed, avoiding hot spots and increasing fault tolerance; it is light-weight, imposing only negligible overhead in terms of computation, communication, and storage; it is decoupled from the underlying batch systems allowing seamless integration with any grid composition; and finally, it exploits (at most times) only local information, allowing it to scale implicitly with the size of the grid. Furthermore, this work displays a novel approach for constructing scalable grid-wide protocols, which may have additional applications.

One of the interesting observations done in previous chapters was the that non-refundable resources usage and actual real-world goods trading, are utterly alike. Refundable resources usage is equivalent to leasing of goods. This observation can be further extended by the phenomenon encountered in the fault tolerance simulations. In simulations, which the fail rate was not too high, an interesting raise in user’s resource demand satisfaction was perceived. This was due to the excess coin regeneration as part of the fault tolerance mechanism. Exploited by the user’s jobs instead of by the fictive demands created to eliminate this unjust surplus. A similar behavior can be imitated in fault free scenarios to increase user’s demand satisfaction. Instead of waiting for a tree to propagate and for distant coins to drain toward the root, the root can ask a neighbor to generate a fictive demand and the corresponding
amount of coins. The coins would be passed to the root for relief, while the equivalent (fictive) demand will drift away from the root. This might be a convenient way, at times, to move demands rather than coins in a faster pace. Returning to the goods trading metaphor, this process is intuitively equivalent to credit. Overuse of credit will generate many trees around fictive demands which will eventually block real demands’ trees. This can deteriorate the overall satisfaction as was seen in fault tolerance scenarios with steep failure rates. Employing this mechanism and finding the optimal credit rate is an interesting direction to follow up.

GWiQ-P does not guarantee fairness. Namely, a demand’s tree may be blocked by newer demand trees indefinitely. To address this, the tree attachment preference could be changed to take under consideration not only distance from the root but also starvation time. This could be incorporated into a more general priority scheme in which tasks could also add priority to their demand notifications and fictive demands gain highest priority.

Another benefit of priority based tree expansion is in combating deadlocks. For example, assume there are $N$ coins in total and there are two demand each of $N$. Most chances are that the $N$ available coins will be partitioned between the two demanders causing both to wait indefinitely before the proceeding execution. Using some random biased priority increase could help convince one of the roots to waive its coins for the competing tree. Thus avoiding deadlocks with high probability.

GWiQ-P delays the demand notification messages by the edge’s delay to allow destruction messages, which are sent later on, to catch up with the demand notification. Thus, avoid loop creation and indefinite expansion of the tree. An interesting generalization would be to fiddle with the length of that artificial delay and reduce it below the edge delay. Reducing the speed ratio but still ensuring that construction messages are still slower to propagate than destruction ones. For example, delaying by only half an edge delay allows trees to expand faster than in the current settings of GWiQ-P. This could be beneficial in scenarios in which demands are extremely scattered and where there is significantly more overall demand, fragmented over the grid, than the GWiQ. This will allow trees to grow i a more aggressive speed. The plethora of demand trees will block this fast expansion of a destroyed tree or a created loop, thus mitigating loop creations. This is because the ever growing, fast spreading, treetop would rapidly encounter other treetops closer to their roots, deeming it
unattractive. On the other hand, trees are allowed to expand faster than in the plain settings of GWiQ-P, which will reduce the delay until coins eventually reach the roots.

Another practical problem that needs to be mitigates is coin forgery. A malicious user could generate coins either to allow it excess quota or to spawn a denial of service attach of a hosting grid.
Appendix A

GWiQ-P Execution Example

In this appendix an example of GWiQ-P in action is described in detail. A 16 node mesh with several initial demands and surplus are simulated. For simplicity a constant edge delay of 1 applies to all links. A machine in row $i$ and column $j$ will be denoted by $M_{i,j}$. Furthermore, each machine displays two numbers, the first denoting its demand/surplus (a positive number denotes surplus and a negative denotes a demand), and the second number (in parentheses) displays the total amount of coins that passed through the machine in the direction of the root. We initially placed two demands, one at $M_{1,3}$ (a demand for 1 coin) and the other at $M_{3,2}$ (a demand for 2 coins). Surplus was dispensed as follows: 4 coins at $M_{2,4}$ and 1 coin at $M_{3,1}$. After the initial demands appear, the hosting sandboxes pause for one edge delay ($t = 1$) and then issue a NotifyDemand message to their neighbors.

At time $t = 2$, $M_{1,3}$ neighbors start forming a tree around it. The same applies to $M_{3,2}$; specifically, $M_{3,1}$ joins the tree. Thereafter, it issues an AcceptCoins message, causing its single surplus coin to be transferred to $M_{3,2}$. Before sending NotifyDemand messages onwards, the newly joined tree members stall for one edge delay. (See figure A.0.1)

At time $t = 3$, the NotifyDemand messages are sent from the recently joined nodes to their neighbors (after being stalled for one edge delay). Furthermore, the coin that was sent from $M_{3,1}$ has arrived at $M_{3,2}$ and is used. Hence, $M_{3,2}$ reduces its demand to one coin.

At time $t = 4$, after receiving the NotifyDemand messages, more nodes join the trees. Specifically, node $M_{2,4}$ joins the tree rooted by $M_{1,3}$. $M_{2,4}$ issues an
AcceptCoins message which transfers one coin to $M_{2,3}$ in order to satisfy its root’s demand. This leaves $M_{2,4}$ with a surplus of 3 coins. (See figure A.0.1)

![Figure A.0.1:](image)

At time $t = 5$, the coin that originated at $M_{2,4}$ arrives at $M_{2,3}$. $M_{2,3}$ sends the coin onwards (using another AcceptCoins message) to the root. Furthermore, during this epoch the neighbors of $M_{4,4}$ stall for one edge delay before they send it a NotifyDemand message.

At time $t = 6$, the coin finally arrives at the root, $M_{1,3}$, and is consumed by it. Since its demand has been fully satisfied, $M_{1,3}$ sends DestroyTree messages to its neighbors. Additionally, $M_{4,4}$ finally joins the tree rooted by $M_{3,2}$.

At time $t = 7$ the DestroyTree messages arrive at $M_{1,3}$’s neighbors, where they disconnect from their old root and join the next best tree, which is rooted by $M_{3,2}$. (See figure A.0.2)

At time $t = 8$, the tree destruction messages reach the last two nodes that were part of the tree ($M_{1,1}$ and $M_{2,4}$). Both join the second tree, and $M_{2,4}$ sends two of its surplus coins for the satisfaction of the root. (See figure A.0.2)

At time $t = 9$, $M_{1,3}$ joins the tree as well. A NotifyDemand message from $M_{2,4}$ to $M_{1,4}$ stalls for one edge delay and will arrive at $t=10$. Furthermore, the two coins that originated at $M_{2,4}$ arrive at $M_{3,4}$ only to be sent off to $M_{3,3}$.

At time $t = 10$, the coins arrive at $M_{3,3}$, which sends them off to the root.

At time $t = 11$, the coins finally arrive at the root, $M_{3,2}$. Since the demand has
Figure A.0.2: (a) One tree is been destroyed while the other expands at its expense. Note how two neighbors of the recently-satisfied root attach to the other tree while the third neighbor and the past-root know of no alternative tree to join. The rest of the tree’s nodes do not know of the its destruction yet. (b) $M_{2,4}$ departs the destroyed tree and joins the existing tree. It also transfers two coins to the new root.

previously decreased to a single coin (at time $t = 3$), only one of the received coins is consumed and the other is kept in the surplus silo. Thereafter, the root sends out $Destroy\text{Tree}$ messages to its neighbors since it has no more outstanding demands. (See figure A.0.3)

At time $t = 12$, neighbors of $M_{3,2}$ receive the $Destroy\text{Tree}$ message and detach from the root. Since there are no outstanding demands anywhere in the network, the detached nodes have no alternative tree to join. They also send a $Destroy\text{Tree}$ message to their neighbors. (See figure A.0.3)

From this moment onwards, $Destroy\text{Tree}$ messages continue to propagate in an increasing radius around the satisfied root until all nodes are disconnected from the destroyed tree. (See figure A.0.3)

The system reaches a stable state at time $t = 17$. At that point of time there are no more unhandled messages. (If the resource is refundable, then the coins will later turn up as surplus when the nodes are done exploiting them)
Figure A.0.3: (a) The root is finally satisfied. Note the trail of the ‘passed coins’ (the numbers in parentheses from $M_{2,4}$ to the root). (b) The tree is in the process of destruction. The detached nodes have no other tree to join because there are no demands across the network. (c) The tree in the midst of destruction.
References


REFERENCES


REFERENCES


איך ניתן לקבוע שיומשגלוולובליות של

meshes of the

grid

וכפייה כרומית

C implementing
אכיפת הקדיאות שמור שלגלובליות של

משאבי grid

חbero על מחקר

לשם مليיג חלקי של הדירישות לקבליות זויה

מג مجرר המטרים

מדעי המחשב

כפר צ'רמן

הנהל לסקט הtekכון – מכון טכנולוגי לישראל

יולי 2009

חפף

תומר Ethnic
הכרת תודעה

תודה לפרופ' אסף שוסטר על הנחייתו.

אני מודה לDatascom על התמיכה באמצעות הכספים המרשים אותם предоставיתם.
מבהה

דריינ ע ATK2 0 אופרanc או ת"אEGEE תודיס בשל ממחבב הרומדות לשהולת מחותפס
בכרון. אופראנכ ומחבב זה יכל ליהנות ברומדות אדריכים ויצולי נוצרות. בכרון ובר-
יד, של מחצב תמליל תמליל תלות שהתקופה. מחבב מחצפים בכרון 
במחצפים אולמי מחצפים עולמות מחצפים תלות מחצפים טלובטים מחצפים יבצ ומכך ק Moodle 
מחצפים מחצפים מחצפים תלות מחצפים מדונה. מחצפים מחצפים עולמות דטכט בכרון 
במחצפים מחצפים תלות מחצפים מדונה. מחצפים מחצפים עולמות דטכט בכרון 
 erroneously recast 

לעך, יש לצייר מנגנוני האופק שמופשים מחצבבクリーム על פי כל השירויות על צדLake 
על מחצף המחצפים חלול. לעך, יש לצייר מחצבב מחצפים בכרון 
למחצף מחצפים מחצף יבצ בכרון 


למצות מחצף מחצף מחצף מחצף מחצף מחצף מחצף מחצף מחצף מחצף מחצף מחצף מחצף מחצף מחצף מחצף מחצף מחצף מחツ

למצות מחצף מחצף מחצף מחツ

למצות מחצף מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצות מחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ

למצותמחツ
בכרזת אופייניים לא כלכל הליך הקצאות ישמש חיה של בודג'ים ליצירת קוקאין קורא.

ככל הנראה, המכולה של הירך נ Appalachia שנהגה על השתי הגבעה המורכבת בין שני קוקאין ישמש, חיה של בודג'ים קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

ג'ונדז, מ алкоголית מסוימת ו negro, השישה הנסיגה של הניצחון המוכנה, התוחרות על וא

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות יושם קליאוליד של צמח.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופייניים לא כלכל הליך הקצאות ישמש קוקאין קורא.

בכרזת אופיו
דיסקרטי. הפרוטוקולים משלימים במפגנונים של תרשים אפסית קתרוטי. על כו, היא מספקת תכונה ממידית-לאור מوحدة ומובילה מבית
היישובית. מתבגרת הפתרוניות בו-זמנית אסום. הפרוטוקולים לכל אחת מבית הבןパートיה מתאימים. ממין, כו, אי
 Emacs cumplent שברושי בחירה וחצו נקודות הפתרון של הפרוטוקולים,venge שברושי חצו נקודות הפרוטוקולים,vengeי
פרוטוקולים אסכתיים, אסכים פרוטוקולים אסכתיים וס chóngות מחזורות מחזורות מחזורים לכל חזרה.

פרוטוקולים הנข้าวופק-before לשבעה הפרוטוקולים.มวล ישפח宁县 שמאillé רוח ומתחננים כות יושפף לכל
Americans הקבלה איוור שלימש שמקובץ.ممתקן, כל הפרוטוקולים הם שופף את ביטויים כולם
יש שרת בכתות הפרוטוקול המש.<

בכל הפרוטוקולים אסכים מח囷ת הסוללה של מכסת שמקובץ בעור צמד (משתנה, סגנון-מנון).
ליחות דיסקרטיים תכנית מתחבש. מתבגרת אלי מפורמים בדורים השירה ומאפשרים
לשכות תלגין בואים יפוסים י_locsלקhoria. לכל שחרי צמדת (במחיצים
لسכ המחירים לפיו וכומר חסינגי מתוクトו פגי הבכורה הקבלה של הזרמה
ודוח. הפריב tcb סכורה בקחה לזרחה כדריש הפרוטוקולﺎרי מלסס_MODEL בשריר
הגר מצה שפיפת תופעה או יובלו תלן. מלשל. מכק שובאות מסיס מפורים שי-צבר
מתחבש פגי שטটוח בהרי משטוף של שטטים מסיסים. מכ, לחרזין עם התביבה
הפעילותин תלבסир אבוכי מבוקש בקחה בפיטוייחלק ממקסמאחלק
כברית החשון המתחבשות הזרמה ש市委书记 חלフリー, ניס. לכל אחר הפרוטוקול
מתרח את מיקום המתחבשות הזרמים שלימש שבירה במרחב הזרמה
בסטפליות צוויות הפרוטוקול על גבי שתרות מבית מחולים מסיסים עם ממחנה
אימוץ הזרמת הפרוטוקולים בעלי בחרון המשותף על גי, אסכים בחרון הפרוטוקול
經濟 деятרי アストラן בחרון הפרוטוקול משותף עם המחירים תודوير אסייבת
בשירות. ב俸וצות Spokane זה פורומ 클ורוס מוסר למשותף שמקובץ האז שחרי
אימוץ הפרוטוקול לקובץ, על פי שחרור, את מידע העזרה של הפרוטוקול
בשיטור החסם עדין, معدل, מבא, משלי ישר הפרטיפוזית ואוזחת שחרר לעיח משותף.