
D C F

A framework for distributed computing

Laurent Flindt Muller
Department of Computer Science
University of Copenhagen
laurent@spiral.dk

Rasmus Resen Amossen
Department of Computer Science
University of Copenhagen
rasmus@resen.org

2004-06-30

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Intention	1
1.3	Achievement	2
1.4	Layout of this document	2
2	General design	3
2.1	Description	3
2.2	Goals	3
2.3	Getting volunteers in contact with projects	4
2.4	Distributing workers among projects	5
2.5	Keeping track of workers	6
2.6	General design overview	7
3	Design of the entities	8
3.1	The worker	8
3.1.1	Connection pattern	8
3.1.2	Function evaluation	9
3.2	The employment office	10
3.2.1	Project management	10
3.2.2	Handling worker requests	10
3.2.3	Handling employment requests	11
3.2.4	Scheduling Workers	11
3.3	The employer	11
3.3.1	The API	12
3.3.2	Distribution and collection of data	12
3.3.3	Security issues	13
3.3.4	Correctness of data	14
3.3.5	Loss of data	14
3.3.6	Worker requests	16
3.3.7	Project administration	18
4	Protocol	19
4.1	Worker to employment office communication	19
4.2	Worker to Employer communication	20
4.3	Employer to employment office communication	21
5	Implementation	26
5.1	Programming language	26
5.2	Worker	26
5.3	The common connection handler	27
5.4	Employment office	27
5.5	Employer	28
5.5.1	Data layer	28
5.5.2	Job layer	30

5.5.3	Work layer	30
6	Testing DCF	31
6.1	Expectations	31
6.2	Strategy	32
6.2.1	Performance	32
6.2.2	Security	33
6.2.3	Real life	33
6.3	Results	34
6.3.1	Performance	34
6.3.2	Security	35
6.3.3	Real life	35
7	Conclusion	37
7.1	Possible enhancements for a future version	37
A	Programmers guide	41
A.1	Managing the project	41
A.2	The API	41
A.2.1	DCF	44
A.2.2	Queue	46
A.2.3	Configuration file	47
B	Package layouts	48
B.1	Header	48
B.2	WorkReq (type 15)	48
B.3	ConToEmp (type 17)	48
B.4	Reconnect (type 16)	49
B.5	EmpReq (type 12)	49
B.6	EmpReqAck (type 2)	49
B.7	EmpReqDen (type 1)	50
B.8	PerformJob (type 4)	50
B.9	Disconnect (type 3)	51
B.10	DeliverResult (type 14)	51
B.11	AddProject (type 8)	52
B.12	AddProjectAck (type 18)	53
B.13	AddProjectDen (type 19)	54
B.14	EditProject (type 9)	54
B.15	EditProjectAck (type 20)	56
B.16	EditProjectDen (type 21)	56
B.17	DelProject (type 10)	57
B.18	DelProjectAck (type 22)	57
B.19	DelProjectDen (type 23)	57
B.20	WorkersReq (type 5)	58
B.21	WorkersReqAck (type 6)	58
B.22	WorkersReqDen (type 7)	59
B.23	SyncReq (type 11)	59

B.24 SyncReqAck (type 24)	60
B.25 SyncReqDen (type 25)	60
C Security test results	61
C.1 File read	61
C.2 File write	61
C.3 Socket access	62
C.4 Start process	63
D Sourcecode	65

List of Figures

1	Statemachine: Worker	22
2	Statemachine: Employer	25
3	Statemachine: Employment office	26
4	Overview of the employer.	29
5	Meaning of estimation variables	32
6	Execution times for $t_e = 0$	34
7	Execution times for $t_e = 30$	35
8	Real life test runs	35
9	A typical ProjectManager session	42
10	A minimal DCF program	43
11	The DataSegment interface	44
12	The Function interface	44
13	The DataSegmentResult interface	44

1 Introduction

This document is intended for readers with some knowledge of computer science in general and more specifically in the areas of networking and distributed computing.

1.1 Motivation

All over the world, a lot of computers are standing idle. The computing power of these computers could be put to more efficient use by allowing others to use their idle CPU time (for example over a network). This is the central idea in distributed computing. A well known example of the concept is the SETI@home project [SETI] developed at the Berkeley University. In this project arbitrary people can install a client and through this client, download a piece of a larger data set, process it locally and return the result to the project home. In case of the SETI@home project the local computation performed is some specific algorithm. Other distributed projects might use the same kind of distribution, but a different computing algorithm.

Instead of each project having to develop their own distributed computing framework, tailored to their specific needs, one could benefit from a general framework providing an abstraction layer for the distribution, evaluation and collection of data. An existing project with this exact purpose is the Berkeley Open Infrastructure for Network Computing (BOINC) project [BOINC], developed at the Berkeley University.¹

1.2 Intention

In this document we aim to design and describe an implementation of a general framework for distributed computing (much in the same manner as BOINC), making it completely transparent for the programmer where the computations are executed. The framework should allow for a programmer to define a function and a partition of a dataset on which the function must be evaluated and then pass these on to the distributed computing framework (through an API) which will handle the distribution, evaluation and collection of data. A helpful person should be able to install a client that will, without his further assistance, get in contact with our distributed computing framework to notify it that is ready to perform work.

There is a very important point in the last paragraph on which the entire design process rests and we will repeat it here for clearness: *Its the programmers responsibility to perform partition of data into data segments which are then processed by the framework* ie. the framework is not responsible for dividing data into smaller units.

The distributed computing framework will be designed for use on the internet as opposed to local area networks or cluster systems. This forces the design to take into account such things as firewalls, network-address-translated (NATed) subnets, clients not delivering back a result and a hole score of security issues related to the untrusted nature of the internet. Still the framework will be able to run on both local area networks and in cluster systems since these are in essence simpler systems than the internet network wise.

Since the framework is designed to work on the internet it must scale well as there are potentially thousands of helpful people out there that wants to share their computing power.

¹While this report was written, SETI@home actually switch their technology to BOINC.

1.3 Achievement

All in all we have successfully achieved all our goals. We have made a working distributed computing framework, that is able to distribute work units to clients, collect them again when they are completed and deliver them back to the programmer.

The system as it stands now could be deployed and used though we would like to continue work to improve it in a version 2 (see section 7.1 on page 37). Setting up the entire system is not totally straight forward at the moment but not hard either (we have not focused on packaging and distribution).

We have not been able to test the system in a real life situation with hundreds or thousands of volunteers and multiple calculation projects, but we have done some estimations on its scalability which shows that it is theoretically able to benefit from having many volunteers if each calculation chunk is relatively time consuming (see section 6 on page 31).

1.4 Layout of this document

The document starts off in section 2 with general design considerations. Here our design goals are specified and the first problems are tackled which leads to an overall design layout of the framework. Next in section 3 we dive more into the different parts of the framework which have been worked out in the previous section. At the end of this section the overall framework design with its different entities will be completed. Following this we describe in section 4 the protocol that is used to communicate between the different entities. With this section completed, the design of the framework is complete.

We then dive into the more implementational details of the design in section 5. This section will give a clear view of how we aim to implement our design.

Following this we devote section 6 to tests. This includes both performance and correctness tests.

Last comes the conclusion in section 7 which holds our thoughts on the design and ideas for improvements.

Appendix A is a programmers guide, describing how the framework is used.

2 General design

In this section the general design of our Distributed Computing Framework, DCF, is discussed. First a list of design goals are presented. From these goals we will sketch our design. A more in-depth design description will be given in section 3 on page 8.

2.1 Description

As described in the introduction in section 1 the aim is to design a distributed computing framework which takes care of the distribution, evaluation and collection of data in association with a distributed computing problem, such that the programmer does not have to think about these issues but can concentrate on creating his algorithm. The aim is *not* to create an MPI ([MPI]) for the internet in the way that the different clients will not be able to communicate with one another but to make a generalized SETI@Home. Also as described in the introduction the data must be split up into the data segments which will be passed out to different clients before the data is passed into the framework. That is, the framework does not divide data that it is passed.

2.2 Goals

We have set some goals for the DCF that our design must meet in order for us to accept it. The keywords for our goals are listed below.

Scalability The DCF must scale well. That is, it should benefit from an increasing number of people that wants to share their computing power. It is also important that an increasing number of volunteers will not result in an accidental flooding of the framework with work requests, resulting in a stall or a complete breakdown.

Security All parts of DCF should be protected from sabotage (whether intentional or not). Volunteers should not be worried that letting foreign code execute on their machines could constitute a security risk and project managers should not be worried that receiving results from volunteers could corrupt their distributed computing project².

Correctness A branch of the security goal above is the need for project managers to be able to trust the validity of incoming calculation results. Validity can pose a problem because some people might think it fun to send back false results and also computers do make calculation mistakes from time to time.

Fair distribution of computing power Different projects have different calculation needs. We would like to distribute computer power as fair as possible among the projects so that projects with large needs does not “steal” volunteers from projects with smaller needs.

Support for NATed subnets In dormitories and other similar places, a lot of computers exists on a private NATed³ subnet. That is, these computers can see “the world” but “the world” cannot see (connect to) them. We need to let these many computers be able to share their computing power with DCF projects. Generally, we would like to allow as many entities of DCF as possible, to reside on NATed subnet.

²From now on referred to simply as a project

³NAT: Network Address Translation

Anonymity In the SETI@home project, volunteers download a small program stub, specially constructed for SETI@home, in order to participate in the project. Other systems, such as BOINC, make volunteers subscribe explicitly to projects of their interest. This way of explicit project subscription has some drawbacks:

- Project managers have to get the volunteers attention before being able to use their resources. This may be hard for small or exotic projects and may even be impossible for a poor student, following a course in parallel programming. In this way, the projects being best at marketing will end up with the largest computing power.
- It may take a serious amount of time from the project is registered, until volunteers discover its relevance. For the student, implementing a week assignment, this may be fatal.

We intent to let DCF be student- and small project friendly in the above sense, by letting the projects be hidden from the volunteers. Instead volunteers should be automatically assigned to projects *as soon* as possible after they request need for computing power. This decision *might* have a drawback: This policy might reduce the incentive for volunteers to share their computing power with the DCF since they do not know what work they are performing, ie the “fun-factor” is reduced and also the work performed could be something of which the volunteer does not approve. We believe that the DCF should be open to everyone and not just the famous. The issues about working for projects of which you do not approve could be resolved by letting the volunteer decide where she gets projects from, so that she knows what group of projects she is working on.

Easy project management As indicated above, it should be fairly easy to add, edit and delete projects and project connection addresses in the DCF network.

Portability Project managers should not focus on making their implementation of function bodies runnable on all possible sorts of volunteer machine architectures, ie. the function should not be compiled for some specific architecture. Instead all code to be distributed should be written either in some sort of scripting language or the compilation should only take place on the actual architecture⁴.

Possibility for isolated execution The DCF system should be constructed in such a way that it can run in an isolated context. E.g. on a closed subnet.

2.3 Getting volunteers in contact with projects

The set of volunteers and project managers resembles the real life set of workers seeking and performing work and employers having work to be done. The volunteers will therefore hence forth be called *workers* and the entities that is in control of a project will be called *employers*.

The first problem that has to be resolved is how to get an employer in need of computing power in touch with workers that are willing to perform work. Because of, among others the *anonymity* and *fair distribution of computing power* goals, we need some centralized way of distributing workers to different employers. There are several possible approaches:

⁴As an example Java byte code could be used or C# intermediate language

1. One centralized register where all projects are registered when they are initially created. The central register would have the responsibility to redirect workers to employers.
2. Several centralized registers working loosely in common where each register has the responsibility for the projects registered in that specific register.
3. Having a range of registers all working in common (like a p2p network) with no fixed registers and where projects may reside in multiple registers.

As we consider the DCF system as a closed entity (or a number of closed entities) we have chosen to have a central register, but still allow for more central registers to exist at the same time, each handling their specific projects. Workers can then connect to such a register and either get forwarded to an employer or another register. A possibility would be for the different registers to share information about worker needs, so that workers could be forwarded from one register to another, but we have chosen not to do this and instead let each register be selfish ie. first guaranteeing workers for the projects that it itself manages before it does any forwarding to other register. These central registers will from now on be called *employment offices* since they put workers in touch with employers.

2.4 Distributing workers among projects

Now that we have a way of putting workers in touch with employers through the employment office we need to find a way to fairly distribute the workers among the projects registered at an employment office. There are two approaches:

1. Volunteers gets a list of registered project contacts addresses (from now on simply called called *contacts*) from the employment office. Now, all the projects on the list can be contacted by the worker, once at a time until one is found that has some work to be done. When the worker reaches the end of the list it can either request a new list from the employment office or begin from the top again. This method has some drawbacks: A small project needing only very few workers could become flooded if a large number of workers has joined the DCF network and are all trying to contact the same project (the project has no way of notifying anyone that it does not need all that computing power). If the number of projects is large, the list may become relatively consuming to download – and may thus result in a high demand for bandwidth at the employment office. Furthermore, depending on how the list is traversed, some projects might benefit from their position on the list and therefor get more computing power.
2. The employment office manages distribution of workers to project contacts. This can be done if the project contacts are also registering their needs at the employment office, ie. it manages how many workers each project wants. Now, the employment office can fairly redirect workers to appropriate contacts and in this way prevent flooding and save bandwidth. A drawback of this approach is that it adds to the complexity of the employment office, since it now has to manage worker requests, ie. keep track of which projects need how many workers, and the projects has to notify the employment office of their worker needs.

We choose the last approach in our design because of the *fair distribution of computing power* goal. Even though it adds to the complexity of the employment office the overall gain in

performance by having control over where the workers end up outweighs this extra complexity. This choice also gives some extra possibilities security wise as we shall see in section 3.3.3 on page 13.

2.5 Keeping track of workers

When it comes to the task of redirecting workers from the employment office, there are immediately three approaches:

1. When a worker comes online it registers at the employment office and then disconnects. Later, when the employment office needs a worker, it contacts one of those that have registered and notifies it that there is work to do. If the worker turns off his/her computer, the employment office could either be send an unregister message or would eventually discover that the worker is no longer online when it tries to connect to it. One advantage with this approach is that it scales well with the number of workers, since the employment office only has to maintain an IP address per worker. There is however a serious problem with this approach, since the employment office has to establish a connection to the worker. This does not allow for the workers to reside on a NATed network.
2. The worker polls the employment office. That is, it connects from time to time to ask the employment office if there is any work available and is then either redirected or told to reconnect at a later time (this time could be fixed or specified by the employment office dynamically). This approach has the advantage that workers can now reside on a NATed subnet but the approach does not scale very well since the employment office may be bugged down if there are a lot of workers trying to connect. Another disadvantage is that the employment office might end up waiting for workers to reconnect (and therefore also the employers) before it can forward any. The first disadvantage calls for setting a long timeout before a worker tries to reconnect while the latter calls for a short timeout so a clever balance has to be found.
3. The employment office manages a pool of recently connected workers and their sockets (ie. the connections are not broken as in the previous approach) so the employment office does not have to reestablish a connection to notify a worker that there is work to do. This allows for the workers to reside behind a NATed network as wanted, but causes some scalability issues: Because of the limit on the number of open sockets the pool can not grow indefinitely and at some point the employment office might have to reject new workers because the pool is filled up.

Even though it seems that the first approach is more scalable than the two others it has the major drawback that workers cannot reside on a NATed subnet. This approach is therefore not an option since this conflicts with one of our design goals.

We have chosen a combination of the two last approaches. That is, we buffer the workers in a pool, but to not completely loose the workers when the pool fills up, the workers are told to retry again at a later time when this happens. This, of course, gives rise to the same problems as the polling approach as the pool might get empty even though there exists non occupied workers out there. But by having a worker pool, the total request for new workers would have to be as large as the pool and occur in very short time intervals in order for this to happen. So the larger the worker pool, the less severe this problem is. With this approach we still lose a bit on the scalable front since sockets needs to be maintained for the buffered workers but

with a good pool size and some intelligent timeouts for worker reconnects it should be possible to minimize the impact of this.

2.6 General design overview

With the above decisions we are now able to sketch a design that meet some of our goals. That is, we have the following three entities:

The employment office becomes the central nerve in DCF. It manages all project data inclusive the need for more workers at project contacts. All connecting workers are placed in a pool, if the pool is not filled up, and otherwise the workers are asked to reconnect again at a later time. The employment office should provide a fair distribution of workers to employers.

The employers provides the framework and API for the project managers (programmers). By using the API, the programmer will be able to enqueue data segments on which to perform a function and will later be able to receive the results of the calculations. The employers task is then to ask the employment office for workers if needed, send data segments and function body to workers, receive results, ensure their correctness and deliver the results back to the programmer, either in the same order as their corresponding data segment was pushed, or by the order they are received.

The workers are contacting the employment office for employment. When they receive a redirect, they connect to an employer. Whether they will stay here forever or if they disconnects at some point, is discussed in section 3.1.1 on the following page. By the anatomy of the DCF design, this distribution of code execution would be the perfect way of spreading harmful code (e.g. a virus), so the worker implementation should encapsulate execution of the function bodies in a secure environment.

In the following sections we will dive deeper into each of these entities and refine our design until we have a complete design that meet all our goals.

3 Design of the entities

In section 2 we developed and outlined a general design of the DCF. In this section we will take a more in depth look at each of the three entities to complete the design.

3.1 The worker

The tiniest entity in DCF is the worker, due to the fact that its only tasks is to request work and perform it. There are two main issues that we have to deal with:

- How and when do we want the worker to connect to the employment office and the employers?
- How do we protect the worker computer against function bodies from renegade employers, that might intent to corrupt the worker?

3.1.1 Connection pattern

The connection pattern of the worker has already been loosely discussed in section 2 on page 3. As mentioned, we intent to let the worker be responsible of establishing all its connections so that it can reside on a NATed subnet (see section 2.5 on page 6). The employment office has the control over which employer the worker connects to.

After having being redirected to an employer (and successfully connected to it), the worker has to wait for further directions from the employer. The employer can either decide to which to fire the worker or give it a work assignment. Getting fired should clearly make the worker return immediately to the employment office, requesting a new employment. Being done with the work assignment and result delivery, the worker could do one of the following:

- It disconnects from the current employer and returns to the employment office for a new employment. Hereby we would achieve the largest possible flow of new workers at the employment office, resulting in a more varying worker architecture (say, machine power). If an employer has been unlucky and mostly gotten slow workers in its request, the large flow of workers would shrink the impact of this unluckiness on the total performance of the employer since it is assigned new workers every time and is thus not stuck with the same batch of “bad” workers. A drawback of this cycling is that it prevents the employer from caching workers in some pool which, due to the fact that the process of getting new workers to be redirected from the employment office is expected to be slow, could result in the employer stalling so often that it would impact of the performance negatively. Also, it would increase the load on the employment office.
- The worker stays at the employer. Now the employer can manage a local pool of workers, we save bandwidth at the employment office, but we got the risk that an employer ends up with a pool of workers which calculation power is strictly less than the market average. Also in an environment with few workers, one or two projects might end up with all the workers while newer projects receive none.

We estimate the pros of the impact of the local worker pool to be larger than the cons and therefore choose the second approach in our design. To reduce the risk of older projects hugging all the workers for themselves the workers will disconnect from an employer if it has

not gotten work to do for a while (this of course only works if the employer has requested to many workers and is not able to occupy all of them).

To recap the connection pattern of the worker: The worker relies on instructions from employment office and employers for all its connection decisions (except for timeouts).

3.1.2 Function evaluation

When the function body has been transmitted to the worker, we must ensure that it gets evaluated in a secure manor in order to protect the volunteer against renegade employers. We have considered some possible security risks and situations that may become annoying for the worker:

Unwanted information gathering An employer could construct a function body that gathered sensitive information stored on the workers computers. Clearly this should not be allowed. Our solution is to restrict the function bodies ability to read sensitive information by making a positive list of permitted reads.

Spoofing The function body could establish connections from the worker, thereby performing possible unwanted actions with the identity of the worker. To prevent this function bodies are not allowed to create any sockets or access the network.

Local system corruption Let S be the worker host system state before evaluation of the function body. We define the function body to be corrupting, if it installs any kind of unwanted software on the worker host or if it modifies or deletes attributes in S in an unwanted manor. E.g. a function body installing a virus, a back door or modifies any system configuration will be defined as corrupting. As the function in the above definition may change some attributes of S without being defined as corrupting (depending on the worker preferences), there is a call for a quite complex resource security mechanism. However, we choose the more simple approach, to limit all function activities to be non-state modifying. That is, we only permit changes of S made by the OS or by side effects, directly related to the code *evaluation* itself. Hereby we especially deny all kind of disk access.

Resource trashing Without corrupting the worker permanently, the function could also just trash all worker resources, in practice resulting in a stalled host. E.g. by doing heavy disk access, filling up the worker disk, filling up memory, spawning several processes or threads occupying system resources for no reason or by using more network bandwidth than wanted. In order to control all resource access used by the function, we must encapsulate its execution in a controlled environment.

Our solution is to disallow all function connection establishments to prevent the function itself from occupying any network bandwidth. The worker program itself might receive a large data chunk for calculation and the function could also produce a large data segment itself – making the worker program occupy the bandwidth. We will solve this by implementing a bandwidth controlling wrapper for all sends/receives performed by the worker so that the worker can indicate how much bandwidth can be used.

After much though we have not found a good way to control disk space usage, memory usage or general CPU resource stealing by spawning many threads. With regards to memory usage and CPU resources this is not a critical issue since this is essentially the

responsibility of the operating system on which the employer runs. With regards to disk usage, we have decided to simply disallow any writing to disk.

To provide the most secure evaluation, we must provide an environment which has at least the union of all restrictions presented above. That is, we deny any disk access, deny any connection establishment, limit reads of system properties as much as possible and implement a configurable bandwidth usage controlling interface to send/receive packages. Notice that our denial of disk access and any permanent state changes will prevent a function to be able to restart from some latest check point in its calculation, if the worker machine is rebooted.

3.2 The employment office

3.2.1 Project management

Employers can add, edit and delete projects at the employment office, but we must somehow ensure, that employers can only manipulate their own projects. This could be achieved by assigning a unique project ID to each project, and keeping this ID secret throughout the projects lifespan. But as we would like to offer future opportunities for non project owners to uniquely refer to a project, we choose to assign a public ID and a longer secret key for each project. Employers then send both ID and key to the employment office in order to manage their projects.

For project data to be valid, we must ensure:

Nonempty project title/description Each project must be assigned a title and a description at the employment office. We choose to force the title to be unique in order to avoid misuse and confusement between registered projects.

Contacts unique All project contacts should be uniquely registered. That is, no two projects must register the same contacts and a project itself must register a given contact at most once.

3.2.2 Handling worker requests

As argued earlier we only permit employers with a valid project ID and a valid key to request workers. To avoid malfunctioning employers to request tons of workers they do not need, we set an upper limit, L , on the number of workers that each project can have in request. We could assign worker requests to projects and redirect incoming workers equally to the registered contacts. But different contacts may have different needs. Therefore we choose to let worker requests be assigned to the contacts. So, let R be the number of workers already requested by a project contact and r be the number in an incoming request. If $r > 0$, we register $\min(L - R, r)$ workers for redirect to the contact.

As we also describe in section 3.3.3 on page 13, we create a key for each worker in a request and send these to the contact to ensure, that each of the incoming workers at the employer has been redirected from the employment office. This prevents a group of potentially malicious people from all connecting to the same employer causing havoc (for example, they all sending back wrong results), since the key ensures that the redirection has occurred through the employment office.

3.2.3 Handling employment requests

As earlier stated, workers connect to the employment office, requesting employment. In section 2.4 on page 5 we argued for managing a pool of workers at the employment office. We keep this decision but add two more perspectives:

- At a worker connect to a no-pool implementation, if we directly redirected it to the next contact that was scheduled for a worker, we would keep the employment office relatively simple. However, we again obtain a problem if no contacts needs a worker when the worker connects: The worker has to disconnect and we are forced to wait until the next worker (re)connects before we can redirect it to a contact if a contact suddenly connects with a request.
- By placing the worker into a worker pool, a separate worker scheduler can take care of distributing the workers. Depending on the pool size and the amount of workers, we may reduce or entirely remove the waiting time necessary to redirect a worker.

3.2.4 Scheduling Workers

The goal of the worker scheduler is to distribute the workers *fairly*. That is, every project should have the same opportunities for getting workers. Also, we want all employers to wait approximately the same amount of time for new workers to arrive. As operating systems does when fairly scheduling processes, we intent to use a round-robin scheduling method. Let P_1, P_2, \dots denote projects and let $C_{i,1}, C_{i,2}, \dots$ denote the contacts for project i . Now, we do the round-robin as described in algorithm 1. In the algorithm, $\min(C)$ returns the contact $c \in C$ with lowest index.

Algorithm 1 Worker scheduling

```

1: while true do
2:   for  $p = 1, \dots, n$  do
3:      $C \leftarrow$  all unmarked  $C_{p,i}$  having at least one pending request
4:      $c \leftarrow \min(C)$ 
5:     Get a worker,  $w$ , from worker pool
6:     Redirect  $w$  to  $c$ 
7:     Mark  $c$ 
8:   end for
9:   for  $p = 1, \dots, n$  do
10:    if all  $c \in C_p$  is marked then
11:      Remove marks from all  $c \in C_p$ 
12:    end if
13:   end for
14: end while

```

3.3 The employer

The employer is by far the most complex entity and has several responsibilities. Many of these have already been described in section 2. The responsibilities are:

Implement the API specified by the DCF The employer must of course implement the interface that the DCF present to the programmer.

Distribute data The employer must distribute the different data segments that the programmer provides through the API to workers, so that the data segments can be processed by the workers.

Collect data The employer must collect the processed data segments from the workers and deliver them back to the programmer (through the API) in an orderly fashion.

Data loss The employer must guaranty that no data is lost ie. all data that has been passed to it must be evaluated.

Correctness of data The employer must somehow give some guarantees as to whether a result is correct or not ie. the *correctness* goal must be achieved.

Security The employer should not be open to malicious attacks ie. it should not be easy for a person or a group of persons to disrupt the functioning of the employer.

Request workers The employer should ensure that there are always workers ready to perform work for it so that the project must not unnecessarily wait for workers ie. it must request new workers from the employment office when they are needed.

In the following discussions a *job* will mean the combination of a data segment and the function to be taken on that data segment.

3.3.1 The API

From the programmers point of view, the employer should be a black box into which she puts jobs, and from which the results of those jobs are returned to her. The way in which the programmer interacts with the black box is specified by the API. Essentially there are two ways in which data can be passed into the black box and returned. Either the programmer wants the results back in exactly the same order as their corresponding data segments where pushed or the order doesn't matter. This view is represented in the API by a continuous queue (results are returned in the same order in which they are pushed) and a discontinuous queue (results are returned in arbitrary order). Each queue is associated with exactly one function. The programmer pushes data segments into the queue and dequeues results in one of the two ways specified depending on the queue type.

3.3.2 Distribution and collection of data

The employer must keep track of the workers that it has been contacted by in order to distribute jobs among the workers. This mirrors the discussion in section 2.5 on page 6 about workers and the employment office and the same arguments apply. We want the worker to be able to reside on a NATed subnet, so we have to keep a connection open with the worker in order to pass it a job. As with the employment office we maintain a pool of workers that are ready to perform work, but there is here no mechanism for telling workers that they should reconnect later if the pool is filled. This is because if such a situation arises the employer has requested more workers than it has jobs to perform and the worker should return to the employment office to be employed elsewhere.

After the worker has been passed a job and performed the required calculations, the result should be delivered back to the employer. There are two possibilities for the connection between the worker and the employer while the calculations are performed. Either we require that the connection should remain open or the worker disconnects and reconnects when it is ready to deliver a result. The first possibility has several problems:

1. The calculations make take several hours or even days, so maintaining an open connection would be a waste.
2. As with the employment office there is a scalability issue, since a lot of connections would possibly have to be maintained.

We therefore choose to close the connection between the worker and the employer when a job has been passed down to the worker. This gives rise to some new issues that must be handled:

1. When a worker delivers back a result some identification process must take place in order for the employer to know to which job the result is associated.
2. A rogue client could deliver back false results pretending to be other workers.

Both these issues fall under the more general security issues and will be treated in section 3.3.3.

3.3.3 Security issues

We now look at a range of security issues connected with the delivery of results from workers. We need to ensure that:

1. The worker that connects and delivers a result is the worker that was earlier passed a job.
2. The worker delivers a result for a job that it was previously passed.
3. The worker does not deliver results for the same job more than once.

To resolve these issues the worker needs to provide some kind of proof that it has been passed a job from the employer. Lets first note that we don't want to use IP addresses in authentication, since multiple workers can have the same IP address and IP addresses can change between data segment retrieval and the delivery of the result (for example if the worker uses DHCP and shuts down its internet connection while it works).

One possibility is to provide the worker with a unique identification, which could be assigned either by the employment office or the employer. The employer could then maintain what job a specific worker had received and therefore was expected to deliver.

This would resolve the security issues including those mentioned at the end of section 3.3.2. The first issue in this section is resolved since the identification number of the worker is unique and known only to the worker and the employer. The second issue is resolved since the worker can not indicate which job it wants to deliver a result for, this is only known by the employer. The third issue is resolved if the entry associating the worker and the job is erased after a job is delivered.

We have however chosen to put in some extra redundancy: Upon connecting for the first time to the employer, the worker is assigned an identification number which is unique to this worker until it stops working for the employer in question. When the worker is passed a job, it is also passed a randomly generated key (which is stored in a pool until it is used or it times out) along with an identification number of the job in question. Upon delivery of the result the worker must provide a working key to be let in. It must then provide its identification number and the identification number of the job it delivers a result for, if these do not match the worker is not allowed to deliver the result. This way the employer can authenticate the worker properly and the result can be delivered. After the result has been delivered the identification numbers that associate the worker with the job is erased such that only one result per worker identification number can be delivered.

3.3.4 Correctness of data

As mentioned we need to somehow ensure that the data that is delivered back to the programmer is correct or at least make it as probable as possible⁵. Depending on the type of project the acceptance of an incorrect results might have dire consequences (we will not state any examples here, but leave it up to the imagination of the reader).

There are two reasons why incorrect results can be encountered:

1. There might be rogue workers out there that would like to ruin a project by sending back wrong result (or they might gain something from it).
2. Computers can, not often admittedly, make calculation mistakes.

The only way to verify a result is a 100% correct is to have a correct result to check against. This is however not a possibility in our case, since if we have a correct result why bother making new calculations? The next best thing to this approach is to have a large amount of results from different sources that are all the same. You could call it correct by consensus. This is the approach that we have adopted for the DCF: The programmer can specify how many equal results he wants to see before a result is accepted as correct. This has the implication that the same job will be performed multiple times by different workers. This of course has the risk of prolonging the total execution time, if the amount of workers are scarce, but given enough workers all these calculations will occur in parallel. Another important issue in connection with this approach is that the same worker should not get passed the same job more than once, since the same error might occur again. In order to prevent this we record the IP addresses (not the IDs) of the workers who have performed a specific job in order to not pass the job to this worker again. This has the risk of blocking out a lot of workers (if they reside on the same subnet) but the risk of this happening is slim. Furthermore, we increase the probability to get segments calculated on non-related hosts if their IP addresses differ.

3.3.5 Loss of data

Data loss occurs if a worker is given a job, but then fails to complete it. The machine on which the worker is running might have been shut down, or some other event is preventing the result to get back to the employer. In this case the employer can not wait until the result

⁵There is no way of guaranteeing this 100% since all the computers in the world could be making the same mistake

gets back since it never will and if nothing is done the project will stall forever waiting for that result. Thus there has to be a way for the employer to detect that a job that is currently being worked on by a worker wont be coming back. With such a detection mechanism in place the employer merely has to reassign the job to another worker.

There are several ways in which the employer could detect a worker failure:

1. The worker could at regular send “I’m alive” messages to the employer so that the employer knows that all is well.
2. Each job could be associated with a timeout, so that if no result has been delivered before the timeout occurs the job is reassigned to a new worker.
3. The employer could check up on workers that were taking a long time compared to some timeout.
4. Each job could be send to a lot of different workers, so that the possibility of them all not sending back a result is diminished.

The first approach gives rise to a lot of network traffic and does not scale very well. The employer could get to a point where all its time is spend processing “I’m alive” messages. This approach also has the security issue that a rogue worker could drag out calculation indefinitely by just sending back “I’m alive” messages but not actually doing any work (this could be resolved with also having a timeout).

The second approach scales better network-wise (and a timeout is needed anyhow for the first approach to work with no security issues). The trouble is how to make a good estimate for a timeout. From project to project the calculation times might vary from a few seconds to many hours or even days. So a fixed timeout is not an option, since a wrong estimation can have serious consequences on performance. If it is to long a project might take longer than is necessary while a to short estimate would result in a lot of unnecessary work to be done, which again would take workers away from necessary work. The best way to estimate the timeout is of course to do the work ie. take a data segment and the function and start the calculations. The key issue to note here is that the timeout is associated with the function and not the jobs. We assume that the function takes about the same amount of time processing each data segment. This is of course not always the case, but if the processing time of each result delivered back from workers are taken into account, in the long run a good approximation should be found. In other words, each time a job is delivered back from a worker, the timeout estimate associated with the specific function is updated. This leaves the problem of how to get the very first estimate before any work has been sent out. It is not enough to presume that a result will be returned and that this processing time can be used for an estimate, since such a result might never be coming (so the project stalls). To resolve this, the function can be executed locally on a data segment. That way it is ensured that a estimation will happen at some point. And since there will properly only be one or at least just a few functions in play per project, this should not bug down the employer. This estimation strategy still has some problems: Some workers might be running on super fast machines while others run on more slow machines. The fast machines will add to the estimate average more often and thus in the long run the more slow machines will be timed out. This is not a serious issue, since a worker that has timed out, should be allowed to deliver a result never the less.

The third approach is not acceptable because that does not allow the worker to reside on a NATed network.

The fourth approach is not acceptable since it does not deliver any guarantees to whether a result will be delivered back.

We have chosen the second approach because it is the approach which scales the best and guarantees that a result will be returned (eventually) for every job. There is though a small scalability problem in this approach since all jobs out there has to be kept in some kind of data structure to manage timeouts, which might be a problem if there are many many workers.

3.3.6 Worker requests

The employer must strive to always have enough workers for the jobs available. Ideally the project should request exactly the amount of workers that it will need to finish the project such that the project never has to stall waiting for workers and that no workers are unused. The employer gets workers by requesting them from the employment office and the employment office then forwards workers to the employer. We need to find a request strategy that comes as close as possible to the ideal situation. In the initial discussion we will always assume that the workers that are requested arrive at some point and that each worker only performs one job. We will later refine the strategy in order to address these two issues.

The first strategy that comes to mind is the following: *Every time a worker is needed for a job, the employer contacts the employment office and requests a single worker. The employer then waits for the worker to arrive before it proceeds.* This however is not a good strategy, since the employer will have to wait every time it wants to send out a new job.

An improvement to this first strategy would be to request some fixed amount, n , every time the previously requested workers had been used. By setting n large enough the employer would not have to wait for new workers that often. This strategy however allows for waste of workers: Let T_{tot} be the total number of jobs in the entire lifespan of a project, then if $n|T_{\text{tot}} - 1$ the employer would request $n - 1$ to many workers.

Another variant of this strategy would be to request fixed amount of workers at fixed time intervals instead, but this strategy has the same problems as the one above.

The ideal would be to just request T_{tot} once and for all and be done with requests. The trouble is, that T_{tot} is not known until the project is completed (or very close to it at least). What is possible though is to iteratively request just the amount of workers that we know will have work to do in a near future. That is: Let T_{cur} be the current number of jobs that the employer knows of and that are not being worked on and let R be the total requested workers that have not gotten a job assigned yet. Then a strategy would be to request $T_{\text{cur}} - R$ new workers when appropriate. This could be when a new job enters the employer, ie. when T_{cur} is updated, or when a job is passed to a worker, ie. when R is updated (which would also change T_{cur}), or at some regular time intervals. This way only the precise amount of workers that are needed will be requested. Requesting new workers every time a job enters the employer is not optimal, since this is just requesting one worker at a time which is the first strategy again. This leaves regular time intervals or when a worker is passed a job. We have chosen to request workers when a worker is passed a job since this ensures that we are not waiting for an request to occur before we can proceed (if the pool gets emptied faster than the timeout for new requests).

We now proceed with this last strategy to refine it with respect to the two assumptions made earlier. That is, we assumed that workers that are requested arrive at some point and that no worker performs more than one job.

Lets start by looking at the first assumption, which consists of two parts

1. There is an unlimited number of workers.
2. A worker that is forwarded from the employment office is never lost.

None of these holds in the real world so we need to handle that there is a limited amount of workers and that workers forwarded from the employment office might fail to reach the employer. Both will result in the pool of workers at the employer being empty. In the first case there is not much to be done though and the employer will simply have to wait till more workers arrive into the DCF network. The second case is a problem though since we are missing a worker and a new must be requested. There are several ways in which this could be handled:

1. The employment office keeps track of the workers that it has forwarded, such that a worker must send back a message when it has successfully connected to an employer.
2. The employer notifies the employment office every time a worker has connected.
3. Both the employment office and the employer maintains a counter on how many requests the employer has made. Every time the employer requests new workers this amount is added to its local counter, and when a worker connects to the employer its local counter is decremented. Similarly the employment office adds new requests to its local counter when they are received from the employer and decrements the counter when workers are forwarded. That way the employer can detect that a worker has failed to reach it if the two counters does not correspond.

The first approach gives rise to a lot of network traffic and constitutes a security risk since a rogue worker could notify the employment office that it had in fact connected to the employer even though it had not. It also requires a lot of bookkeeping on the part of the employment office.

The second approach also brings about extra network traffic and gives the employment office a lot extra work since it must process incoming worker received notification from every project it handles.

We therefore choose the third approach. In this approach the problem is reduced to keeping two counters (one in the employer and one in the employment office) synchronized. This might not seem a simplification, but the point that makes it worthwhile is that this synchronization only has to take place when something goes wrong. If no workers are lost synchronization would never have to take place and thus in a well functioning network no extra network traffic ensues. This synchronization mechanism also allows the employer to detect that no more workers are available in the network, since if the employer and employment office counter has the same value and the employers pool of workers is empty it must mean that no more workers are available globally. This leaves the decision of when to trigger synchronization. We have chosen to trigger synchronization when the pool of workers is empty and to synchronize whenever the employer and employment office speak to each other.

Lets now look at the last assumption made, that each worker only performs one job. That is of course not very optimal, since after a worker has delivered a result for the first job, it might as well be given a new instead of going back to the employment office and being redirected again. This allows for one employer to accumulate a lot of workers, but since the workers will return to the employment office if they get no new work for a period of time, this should not be a big issue. The question is now whether to take these active workers ie.

workers that are currently working on a job (a job that has not timed out), into account when performing the calculation of how many new workers to request. One could argue that these workers are irrelevant since there is no way of telling when or even if they will be back. But not taking them into account could possibly lead to too many workers being requested. We have chosen to take the number of currently active workers into account, but to scale down their importance. That is one active worker does not count as a hole worker when requesting new ones, since there is a possibility it will not return. The expression for requesting new workers now looks as follows: Let T_{cur} and R be given as earlier, let A be the number of active workers and let $0 < k < 1$ then we request $T_{\text{cur}} - (k \cdot A + R)$ new workers.

So to recap this section we request workers every time we pass a job to a worker. The amount we request is given by $T_{\text{cur}} - (k \cdot A + R)$. If the pool of workers at the employer at any time gets empty this is a signal that something is wrong and at this point the employer and the employment office synchronize their respective counters of how many requests the employer has made. This enables the employer to detect if requested workers that the employment office has forwarded have been lost.

3.3.7 Project administration

The last topic to be addressed is how to manage a project. There are three parts to managing a project: Creation, editing and deleting. One could argue that project administration is not really part of the DCF but something that has to be done on the side which is essentially true, ie. project administration is not something that should reside in the API. Basically the employment office should implement some kind of mechanism for adding, editing and deleting projects. This mechanism could then be managed by some agent who could be a person receiving emails from project managers and then reflecting the changes in the employment office, a web server interacting through a web page with project managers and though a back end with the employment office or it could be some custom network server.

We have chosen to let the mechanism in which way the employment office gets told to create, edit or delete a project be the receipt of network packages. That is, we have essentially build in a small custom server into the employment office which handles management of the projects through special packages. The sender of these packages could be essentially any of the above agents or some third party program.

4 Protocol

In this section we will discuss the network protocol that will be used for communicating between the different entities. We have chosen to use TCP as our means of connection as opposed to UDP. This choice resides in the fact that TCP connections constitute two-way communication channels while UDP does not. The reason we need two-way communication, is because we want workers to be able to reside on NATed networks.

We have chosen to use packages as the way entities are communicating. These packages are not fixed size, but can vary in length. Each package represents a certain action that we want to perform, such as “Do this work” or “I want more workers”. Most packages, but not all, needs to be acknowledged by the receiver. All packages has the same header which consists of:

- Package Version.
- Package Type.
- Package length.

The layout of the header and all other packages can be found in appendix B on page 48. In the description of the different packages these fields will not be mentioned, as they occur in every package.

In the following section we will first present the worker-to-employment-office communication, then the worker-to-employer communication and end with the employer-to-employment-office communication.

4.1 Worker to employment office communication

The communication between the worker and the employment office is rather simple. The worker needs a way to notify the employment office that it is there and ready to be forwarded to some employer. This is done with the **WorkReq** package which contains:

- A last timeout entry (in milliseconds) which is used by the employment office to calculate how long this worker should wait before reconnecting if the employment office pool of workers is filled. The initial value should be 0.

Upon having send the **WorkReq** package the worker expects to get told to connect to an employer or to either reconnect at the same employment office again or try another one. These two events are represented by the **ConToEmp** and **Reconnect** packages respectively. The **ConToEmp** package contains:

- The key the worker must provide in order to be let into the employer.
- The IP address and port number of the employer to be contacted.

The **Reconnect** package contains:

- An action field which specifies which action is to be undertaken.
- The timeout before the action should be undertaken.

- The IP address and port (if applicable) of where the action should take place.

The action field can take one of the following values:

1. Reconnect to the given IP and port after waiting the timeout specified.
2. Reconnect permanently to the given IP and port after waiting the timeout specified. Permanently means that this worker should use this new IP and port as the default when connecting at a later time.

Upon receipt of any of the two packages the worker breaks the connection with the employment office.

4.2 Worker to Employer communication

Communication between worker and employer is also rather simple. The worker needs to notify the employer that it is here and ready to work. This is done with the **EmpReq** package which contains:

- The key the worker received from the employment office, and which it should use to enter the employer.

After having send the **EmpReq** package the worker expects either an **EmpReqAck** or an **EmpReqDen** package depending on whether the worker is employed or not. The **EmpReqAck** contains:

- The worker identification number that this worker is assigned.

The **EmpReqDen** package contains:

- The reason why the worker was not employed.

The reason can take on the following values:

1. A wrong key was supplied.
2. To many workers are already present at the employer.

In both cases the worker falls back to its initial state and tries to connect to the employment office again.

After the receive of an **EmpReqAck** package, the worker expects either some work to be done or to be fired from the employer (that is, the employer no longer needs its services). These two events are represented by the **PerformJob** and **Disconnect** packages respectively. The **perform job** package contains:

- The key to be provided when the worker wants to return the result.
- The job identification number of the job the worker is working on.
- The function and data segment that this job consists of.
- The **.jar** file which contains the classes representing the function and data.

After the receive of this package the worker closes the connection with the employer and starts working. If the worker receives any other package it disconnects and falls back to the initial state where it tries to contact the employment office.

The other package the worker can receive instead of the `PerformJob` package is the `Disconnect` package. This package contains):

- The reason why the worker should disconnect.

At this point the reason can only take on the value 8 which means that the worker was fired. Upon receive of an `Disconnect` package the worker disconnects and returns to its initial state where it tries to contact the employment office.

Upon having finished work on the job the worker received, the worker must return the result to the employer. This is done with the `DeliverResult` package. This package contains:

- The key it received when it was passed the job.
- The job identification number of the job received.
- The result of the job.

After having delivered this package the worker falls back to the same state as when it had just connected to the employer. But since it has just delivered a result the reasons for a disconnection via the `Disconnect` package are more numerous:

1. The package was not the expected package.
2. The package was not the correct version.
3. The package was too big.
4. There were too many workers in the pool for the worker to be reemployed.
5. The key provided was incorrect.
6. The package layout was not correct.
7. The worker had not been assigned this job.

This completes the communication between worker and employer. A complete state machine for the worker can be seen in figure 1 on the next page.

4.3 Employer to employment office communication

All communication between the employer and employment office are request-reply based. The Employer makes a request for some action to be taken to which the employment office makes a reply. After this reply the connection is terminated. There are five types of requests:

1. Request to add a project.
2. Request to edit a project.
3. Request to delete a project.

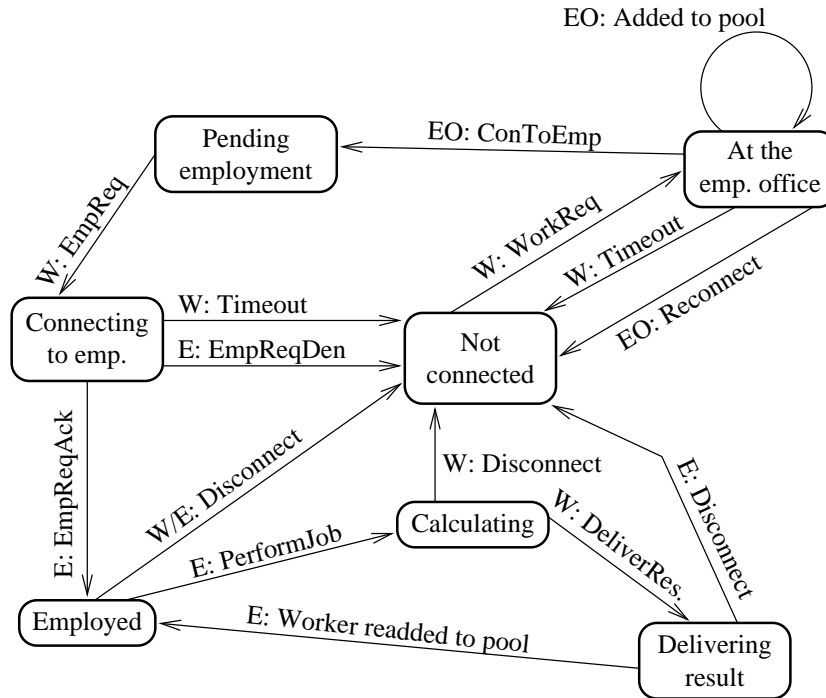


Figure 1: State machine for the worker. The notation at the arrows is “Performer: Action”, e.g. “E: EmpReqAck” means that the employer sends a `EmpReqAck` package.

4. Request for more workers.
5. Request for synchronization.

To each request the employment office can either send back an acknowledgement that the request was executed or a denial meaning that the request was not executed.

The request for adding a new project is represented by the `AddProject` package, which contains:

- The title of the project.
- The description of the project.
- The contacts that should be associated with this project i.e. IP addresses and port numbers that forwarded workers should contact.

In response to this package the employment office can send either a `AddProjectAck` or a `AddProjectDen` depending on whether the requested action was executed or not (here the action was to add a project). The `AddProjectAck` package contains:

- The project number identification and key that the employer must use later on the authenticate itself with the employment office.
- Identification numbers, IP addresses and ports for each of the contacts that were present in the `AddProject` package.

The `AddProjectDen` package has contains:

- The reason why the requested action was not taken.

The reason can take on the following values:

1. That a host was invalid, ie. the host name could not be resolved by the employment office.
2. There was an error in the package.
3. The title was empty.
4. The title given was not unique, ie. there is another project with the same title.
5. Some internal server error occurred.

The request to edit a project is represented by the `EditProject` package, which contains:

- The project identification number and key.
- The new title and description (the title and description will always be set to these in the employment office).
- The identification numbers of the contacts that should be edited and their new IP addresses and port numbers.
- The identification numbers of the contacts to be deleted.
- the IP addresses and port numbers of new contacts to be added.

In the same manner as before the employment office can in response to this package send either an `EditProjectAck` or `EditProjectDen` package. The `EditProjectAck` package contains:

- The identification number, IP addresses and port numbers for each of the new contacts from the `EditProject` package.

The `EditProjectDen` package has the same layout as the `AddProjectDen` package. The reasons for denial are here:

1. Project identification number and the key provided did not match.
2. There was an error in the package.
3. The title was empty.
4. An internal server error occurred.
5. The title given is not unique ie. there is another project with the same title.
6. One of the contact identification numbers provided was invalid.

7. One of the contacts already exists ie the same IP and port.

The request for deleting a project is represented by the **DelProject** package, which contains:

- The identification number and key for the project to be deleted.

As before the employment office can either respond with a **DelProjectAck** or a **DelProjectDen** package. The **DelProjectAck** package is empty and is merely a reply.

The **DelProjectDen** package has the same layout as the **AddProjectDen** package. The reasons for denial are here:

1. There was a error in the package.
2. The project identification number and key provided did not match.
3. There was an internal error in the employment office.

The request for more workers is represented by the **WorkersReq** package, which contains:

- The identification number and key for the project in question.
- The amount of workers requested.
- The identification number of the contact that is requesting these workers.

As before the employment office can respond with either a **WorkersReqAck** or a **WorkersReqDen** package. The **WorkersReqAck** package contains:

- The keys that the workers are expected to provide when they try to connect to the employer.
- A timeout for the keys.
- The number of requests for this project currently known to the employment office (this is used for synchronization purposes).

The **WorkersReqDen** package has the same layout as the **AddProjectDen** package. The reasons for denial are here:

1. The project identification number and key provided did not match or the contact identification number was invalid.
2. There was an error in the package.
3. There was an internal error in the employment office.
4. If the number of requested is less than one or if the total number of requests already registered by the employment office exceeds some limit.

The request for synchronization is represented by the **SyncReq** package which contains:

- The identification number and key of the project in question.

- The identification number of the contact that is performing the request.

As before the employment office can either respond with a `SynReqAck` or a `SynReqDen` package. The `SynReqAck` package contains

- The number of requests this contact has on the employment office.

The `SynReqDen` package has the same layout as the `AddProjectDen` package. The reasons for denial are here:

1. The project identification number and key provided did not match or the contact identification number was invalid.
2. There was an error in the package.
3. There was an internal error in the employment office.

This completes the communication between employer and employment office. A complete state machine for the employer and employment office can be found in figure 2 and figure 3 on the next page.

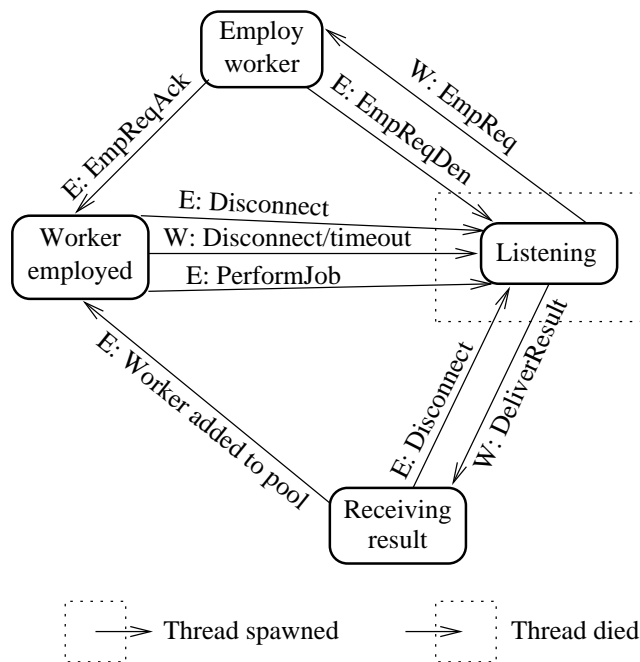


Figure 2: State machine for the employer. As the employer is multi threaded, the diagram should be considered as the lifespan of a socket.

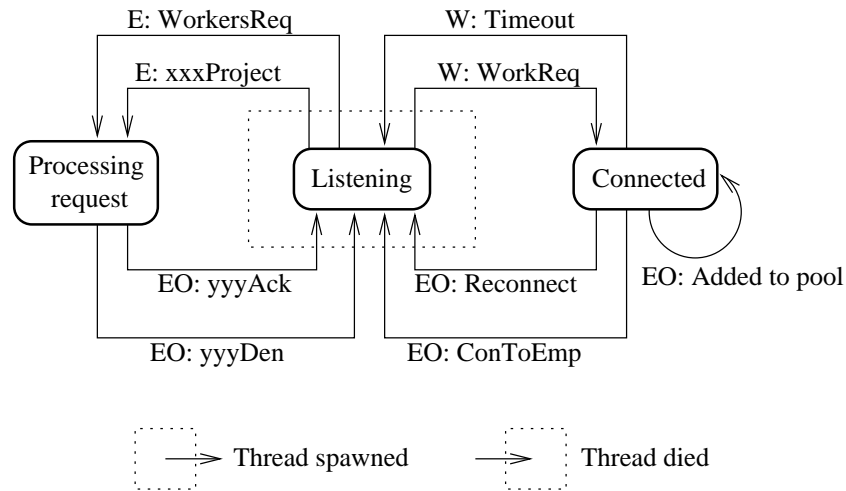


Figure 3: State machine for the employment office. As the employment office is multi threaded, the diagram should be considered as the lifespan of a socket. The “xxxProject” arrow text represents all project mangeling packages while “yyyAck” and “yyyDen” represents all acknowledge- and denial packages from the employment office.

5 Implementation

In this section we give a short implementational description of the three entities and the system as a hole.

5.1 Programming language

The first choice to be made is always which programming language(s) to use. We have chosen the Java programming language. The main reason for this is its portability. This is specially because we want to be able to distribute data and function to workers that might reside on different machine architectures. Had we chosen C/C++ or some language that needs to be compiled for a specific system, the function would have to be either compiled for that specific machine architecture before sending it out to a worker or it would have to be compiled on the workers machine. With Java the worker just needs the Java runtime environment. Another important feature about Java is that it allows instantiated objects to be *serialized*, passed over a network and to be *deserialized* at the receiving end. There is of course a speed tradeoff, since Java is essentially an interpreted language, but we have deemed ease of function distribution a more important issue than speed. Another possibility would have been some kind of scripting language for function bodies, but this would have to be interpreted as well so does not present an advantage over Java. Another alternative to Java would have been C#, but since this language resembles Java a lot and we both had a much better knowledge of Java this last language was preferred.

5.2 Worker

The worker design is quite straight forward and is pretty much a naive implementation of the state machine depicted in figure 1 on page 22. The most interesting part of the worker

implementation is the security handling. That is, we must ensure that the worker doesn't get damaged when reconstructing the data segment and function if the employer has overwritten the unserialization methods, and we must ensure that the function body will not damage the worker either. Lets consider the last issue first.

Java has a convenient mechanism for handling resource permissions, named the *security manager* [SECURITY]. The security manager is controlled by giving a set of policies, which are permissions for resource usage (file system, network access, etc.). We have chosen to rely on this mechanism and the task has then been to encapsulate as much of the employer related handling as possible into a security manager controlled environment. So, when a new job is received, we start a new process, named *secure box*, controlled by the security manager with an empty policy set (which means no permissions at all). We then perform the function calculations inside the box, thereby obtaining the controlled evaluation as desired.

To reconstruct the employer specific data we stream the still serialized data into the secure box. Inside the secure box, the data segment and function objects are unserialized, thereby eliminating the risk of harming the worker with a bad overwritten unserialize method. Similarly, the result set is serialized inside the secure box and streamed to the main process when calculations has finished. The serialized data is then used directly in the package construction without further mangling.

As the serialization of objects does not include any class definitions, the class definitions need to be passed to the worker by the employer. These class definitions are passed along in a .jar file in the package that the worker receives from the employer when receiving a new job.

5.3 The common connection handler

The naive way of managing incoming connections is by handling them as they arrive, one by one. That is, when an incoming connection is being established, all work related to the connection is performed and the process starts listening to the socket again hereafter. This approach is rather inefficient as it prevents the connection related work from being done in parallel and thus may be unnecessarily blocking for new connections.

We have therefore chosen to implement a light *connection manager* thread. The connection manager listens to the socket and spawns a new thread with an appropriate connection handler at each new connection. In that way, the connection manager can quickly return to the socket again and listen for new connections.

We can limit the connection manager to allow a certain maximum number of simultaneous connections and the connection manager can be configured to only receive data at a certain transfer rate.

The connection manager mechanism is used by both the employers and the employment office.

5.4 Employment office

As the employment office plays a quite central role in DCF, we aim to let it be able to restart properly after an eventual interruption, e.g. a crash or a reboot. For example, we don't want to loose track of project data or pending requests. Therefore we cannot just keep information in memory as this may be reset. We need to store data on disk using a mechanism that provides transaction atomicity (a series of actions used for performing a high-level task should be considered as a single task), consistency (only valid, and not partially build, data is stored),

isolation (no two concurrent transactions must interfere with each others execution), durability (restoration of all stored data during, eg. a crash). Luckily those demands⁶ are exactly the purpose of modern database systems. We will therefore use a database for storing project data and contact requests.

Querying a database might be relatively time expensive. Therefore we may speed up performance if we minimizes the amount of queries. To save database access we define a round-robin round trip for the worker scheduler as

1. A single query, performing line 9-13 in algorithm 1 on page 11.
2. A single query, fetching all c 's in line 2-4.
3. For all c 's, a query removing a request when the contact has been forwarded a worker.

Now, when an employer performs a request of n workers, the employment office calculates n keys and returns these to the employer together with a key timeout. This timeout must of course be affected by the round trip time, T , as each project gets at most one worker per round trip. We therefore calculates the timeout, L , as $L = Tnk$, where k is some scalar.

5.5 Employer

The employer is a quite big entity so we have decided to try and layer it. That is to divide it into different layers, one resting on top of the next with the API at the very top. The reason for this is that it makes the employer more manageable and enables us to test different layers separately and to switch layer implementations without having to change the other layers. Without considering the API layer the employer consists of three layers each reflecting a specific functionality in the employer. These layers are the **Data**layer, the **Job**layer and the **Work**layer which will be described in the following three sections (a diagram of the employer can be seen in figure 4 on the following page).

5.5.1 Data layer

At the top is the **Data**layer. This is the layer with which the API communicates. Data segments and their associated function are passed into this layer from the API queues. The responsibility of the **Data**layer is to keep track of the data, and decide when the result associated with a data segment can be delivered back through the API queues to the programmer. Ie. it keeps track of how many equal results has been seen for a specific data segment and decides whether a new evaluation of a data segment is needed. Based on how many equal results must be seen, jobs are created for the data segment and associated function in the job layer. To be more precise the number of jobs that are initially created per data segment equals the number of equal results that must be seen in order for the job to be completed. When a result for a job is received, the job is rescheduled if the following expression is true: Let n be to total number of equal results that must be seen, let n_{\max} be the maximum amount of equal seen results so far and let n_{sched} be the amount job jobs that are scheduled for evaluation. Then the job is rescheduled if $n - n_{\max} > n_{\text{sched}}$.

⁶named ACID

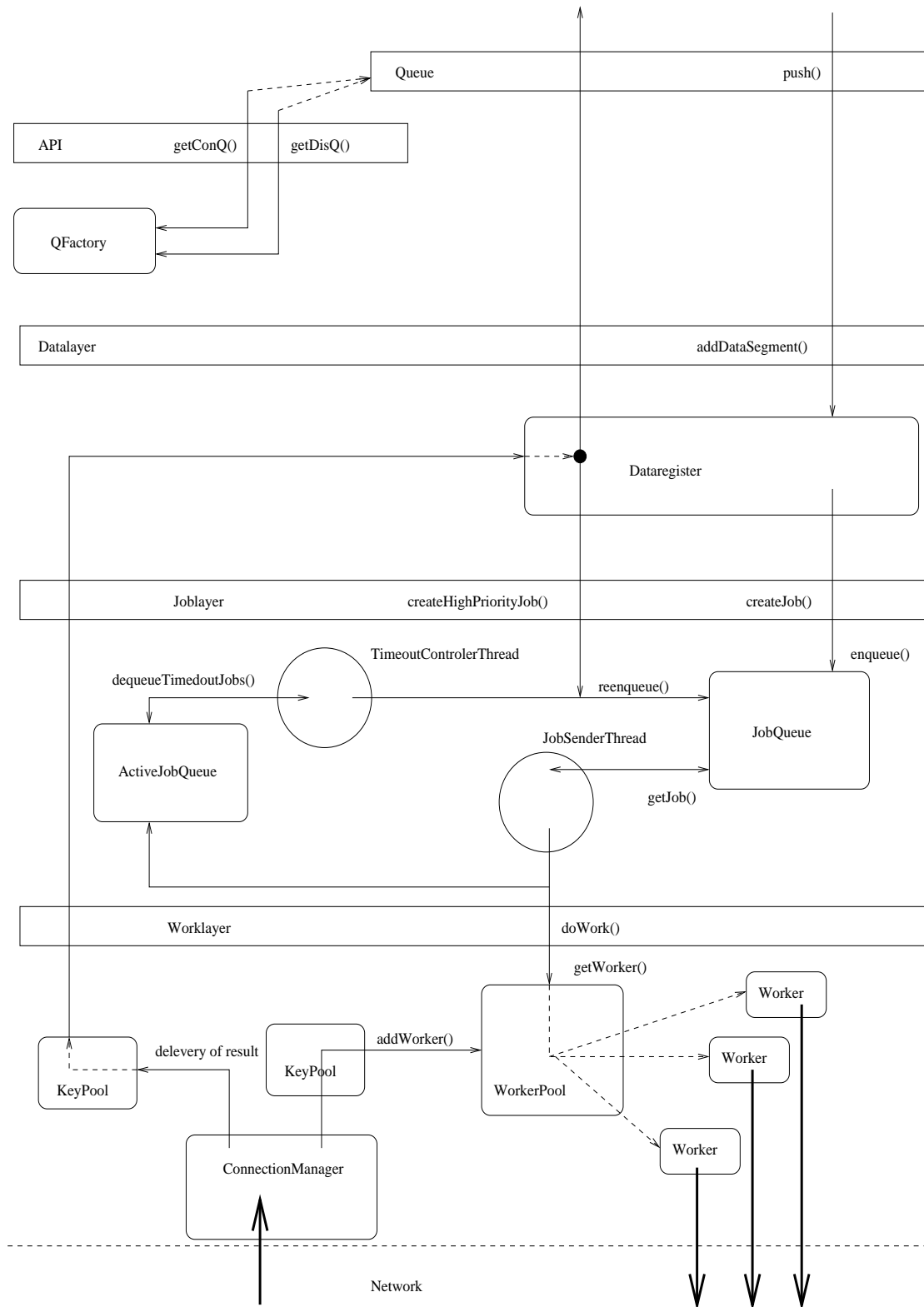


Figure 4: Overview of the employer.

5.5.2 Job layer

Below the **Datalayer** lies the **Joblayer**. The **Datalayer** creates jobs in the **Joblayer**. The **Joblayer** can create a normal priority job or a high priority job, depending on the need of the **Datalayer**. High priority jobs are treated before normal priority ones. The creation of high priority jobs is used by the **Datalayer** when reevaluation of a data segment is needed. This need arises if a result is delivered that is different from the ones seen so far and thus a new result must be evaluated before the wished amount of equal results will be seen.

The **Joblayer** has the responsibility of scheduling jobs for execution in the **Worklayer** and keeping track of which jobs has timed out and must therefore be resubmitted to the **Worklayer**. To this aim, it maintains two data structures: A **Jobqueue** which contains the jobs to be scheduled (there are in fact two queues: One for high priority jobs and one for normal ones) and an **Active Jobqueue** which contains jobs that has been passed to the **Worklayer** and not been delivered back yet. This last data structure is responsible for timing out jobs that have taken to long and reinserting them into the **Jobqueue** (as high priority jobs). It is also in this structure where estimation of execution time takes place.

When the **Joblayer** receives back the result of a job from the **Worklayer** it updates the time estimate for that job, removes the job from the queue where it resides and passes the result back up to the **DataLayer**.

5.5.3 Work layer

The **Worklayer** is the thickest layer. Here resides everything that has to do with the actual evaluation of the jobs ie. establishing contact with workers and the employment office. Also this layer has the responsibility for security.

The **Worklayer** is passed jobs from the **Joblayer** and must then find appropriate workers to which the jobs can be send. In association with sending of jobs, a key must be produced and stored such that the worker can be authenticated when it returns with a result. The **Worklayer** must continuously request new workers from the employment office as described in section 3.3.6 on page 16.

To its aid the **Worklayer** contains two data structures. A **Workerpool** that holds unemployed workers and a **Keypool** into which new keys are generated and from which keys can be removed when they are used. In fact there are two keypools, one holding keys received from the employment office and one holding locally generated keys that workers must provide in order to deliver results for jobs.

The **Worklayer** also manages incoming connections, which can either be new workers that must be placed into the worker pool or a worker returning a result. In both cases the worker must provide a valid key after which the worker is added to the worker pool.

6 Testing DCF

In order to test DCF, we have defined a set of properties we want to measure:

- What is the *overhead* of the framework? That is, if it takes the time, $t = t_d + t_f$, from a data segment is enqueued till its result becomes available, how much time, t_d , is then used by DCF and how much time, t_f , is used by the function to perform calculation.
- How good does DCF handle *parallel* evaluations? That is, will n uniform workers perform half as good as $2n$?
- Have we succeeded in our *security* implementation, encapsulating the function calculation properly?
- How does DCF perform in a real life situation?

6.1 Expectations

We aim to find a theoretical execution time for a DCF project. So let n_w and n_j be the number of workers in pool and jobs to be performed, resp. Also, let o_{emp} , o_w be the overhead per job at the the employer and the workers, resp. and let o_{eo} be the time it takes the employment office to redirect a worker to the employer. The total amount of time spend at the employment office is then approximately $n_w o_{\text{eo}}$ and the time spend at the employer, $n_j o_{\text{emp}}$. If we let t_e denote the average execution time of the function body and t_x denote the average transmission time of data between worker and employer, we can formulate an expression for the total execution time, T :

$$T \approx n_w o_{\text{eo}} + n_j o_{\text{emp}} + \frac{n_j}{n_w} (o_w + t_x + t_e) \quad (1)$$

This is valid in an optimal execution environment where no jobs has to be reevaluated because of timeouts or because of wrong received results. Also, as described in section 3.3.4 on page 14, increasing the required amount of equal results, will also increase the number of jobs. Therefore, as n_j is determined partially from this amount, we will not need to consider this separately. There are several uncertain parts of the above estimation:

- The employer might start delegating jobs to workers before all of them has arrived. Therefore $n_w o_w + n_j o_{\text{emp}}$ should be considered as an upper limit for the employment office / employer local time consumption.
- The $t_x + t_e$ can vary a lot. If t_e is large, t_x might become negligible and vice verse. Depending on the size of the data to be transmitted, it becomes important that the employer has a lot of bandwidth.
- As stated, workers might fail to calculate or they might time out, forcing the job to be rescheduled. A job reschedule is equal to increasing n_j with 1.

An estimation for the pure DCF framework overhead can be found by letting $t_x = t_e = 0$. Hereby we see that the two major factors in the execution time is the employment office and especially the employer, as n_j may become very large.

So, how do we expect DCF to perform in parallel executions, scaling the number of workers with a factor $k > 1$, where we restrict that $kn_w \leq n_j$, as workers cannot share a job execution?

Let T_k denote the total execution time, for the scaled amount of workers. The execution time, relatively to T is then

$$\frac{T_k}{T} = \frac{kn_w o_{eo} + n_j o_{emp} + \frac{n_j}{kn_w}(o_w + t_x + t_e)}{n_w o_{eo} + n_j o_{emp} + \frac{n_j}{n_w}(o_w + t_x + t_e)}. \quad (2)$$

We can now consider the properties of an execution time decrease, that is $\frac{T_k}{T} < 1$:

$$\frac{T_k}{T} < 1 \Rightarrow (k-1)n_w o_{eo} < \left(1 - \frac{1}{k}\right) \frac{n_j}{n_w} (o_w + t_x + t_e) \quad (3)$$

This inequality tells us, that we should expect a decrease in the total execution time if the time it takes to request the additional workers is less than the time we win by performing the parallel execution with the additional workers – a relatively trivial statement, one might say. Notice, that the employment office overhead is not important here.

Lets look at what should theoretically happen with the fraction $\frac{T_k}{T}$, if we keep all variable static, except one, and let the one increase infinitely. A table containing the variables and the limits is seen in figure 5. As seen in the tabular, we should expect a k times increased

Variable	Limit
o_{eo}	k
o_{emp}	1
o_w	$1/k$
n_w	$1/k$
n_j	$\frac{ko_{emp}n_w + o_w + t_x + t_e}{k(o_{emp}n_w + o_w + t_x + t_e)} = \frac{k\alpha + \beta}{k(\alpha + \beta)}$
t_e	$1/k$
t_x	$1/k$

Figure 5: Letting the variables in equation 2 grow we get some relative speedup limits for adding more workers. For n_j we get a fairly complex fraction, but as everything, except k , is constants we can replace the constants with α and β and obtain a simpler fraction. We see, that the amount of speedup is determined heavily by β .

execution time, as o_{eo} grows whereas all other variables seems to benefit from more workers, as they grow. Especially, as the number of jobs, n_j , increases it becomes more important with larger and more consuming execution chunks, if the project should gain from more workers – exactly as concluded from equation 3.

6.2 Strategy

6.2.1 Performance

We will try to measure o_{eo} , o_{emp} and o_w by doing a series of tests. For this purpose, we have constructed a function that does not calculate anything, but for which we can adjust the following parameters:

EqualsToSee The amount of equal results we want to see.

Local execution time Time consumption of the function at the employer host. This is implemented as making the process sleep, so this is independent from host performance.

Worker time, lower limit A factor $0 < l \leq 1$ determining the lowest possible worker execution time.

Worker time, upper limit A factor $k \geq 1$ determining the highest possible worker execution time.

Dead worker probability The probability for the worker to die (and never return).

Wrong result probability The probability for the worker to deliver a wrong result.

Segments in queue Number of simultaneously pushed data segments.

Total segments The total number of segments to be pushed and received for this test.

Our test project then just pushes the specified amount of jobs. The test itself consists of two runs. During the first run no workers exists at the employer so time is spend requesting and getting these workers from the employment office. During the second run, the workers are already cached in the pool, thereby eliminating the o_{e0} factor.

For the performance issues, we make two series of test runs with different amount of workers and jobs. In the first series, we set $t_e = 0$, let the worker count, n_w , be 1, 2, 3, 8, 16 and fix n_j to 64. This makes o_{emp} and $(o_w + t_x)$ our variables to find, as we know n_j and n_w . In our test environment all worker hosts are located on the same LAN. Our `PerformJob` package size is 16955 bytes and the `DeliverResult` package size is 192071 bytes, which is 209026 bytes in total. On a 100 mbit not congested LAN, this transmission should take no longer than 0.02 seconds, so we can assume this is 0 for these tests. From the first test series we can now pick two worker/job count configurations from the second runs (with the workers cached in pool) and solve two equations with two unknowns, to find an estimate for o_{emp} and o_w . Next, we can pick a configuration from the first runs, to estimate o_{e0} .

Having the three estimations we will try to predict the running time for 8 and 16 workers with $t_e = 30$ seconds.

6.2.2 Security

For the security question, we construct three functions, each trying to perform one of the following actions:

- Access a file (read and write)
- Access a socket
- Start a new process

In order for the test to succeed, the worker must detect the security violations and disconnect from the employer. Thus the employer shouldn't receive any results.

6.2.3 Real life

For the real life test, we will test the frameworks ability to handle wrong incoming results and its tolerance for different worker speeds. We decided not to spend time on finding and solving a real problem as this would be run in our local LAN anyway (which are not representable for the real world outside). Instead we will use the same parameterizable function as in the earlier performance estimation tests but with more stressing parameters.

6.3 Results

6.3.1 Performance

We have for the test series made three test executions of every n_w configuration. The average execution times is listed in figure 6. As we predicted in section 6.1 on page 31, the employment

n_w	n_j	1. run	2. run	T_k/T
16	64	27.4	22.4	0.9
8	64	31.1	25.9	0.7
4	64	42.2	38.8	0.6
2	64	63.4	62.1	0.6
1	64	109.6	108.7	-

Figure 6: The tabular shows the average execution times for the 5 tests having $t_e = 0$ and at each row $k = 2$. It is seen that $1/k \leq \frac{T_k}{T} < 1$. If the employment office should have made a large impact, $\frac{T_k}{T}$ would have been closer to k .

office time consumption has only minor impact on the execution time: If the employment office overhead was large, relatively to all other variables static, $\frac{T_k}{T}$ would have approached 2, as $k = 2$, but instead it is closer to $1/2$.

We will now estimate o_{emp} and o_w . This is done by considering equation 1 on page 31 and looking at the data in the rows with $n_w = 1$ and $n_w = 16$, 2. run. We can then solve the equations

$$\begin{aligned} 0 + 64o_{\text{emp}} + \frac{64}{4}(o_w + 0 + 0) &= 22.4 \\ 0 + 64o_{\text{emp}} + \frac{64}{1}(o_w + 0 + 0) &= 108.7 \end{aligned}$$

as we reminds the reader, that we assume $o_{\text{eo}} = t_e = t_x = 0$ for this test, and get

$$o_{\text{emp}} \approx 0.3 \quad \text{and} \quad o_w \approx 1.4. \quad (4)$$

One could probably find more accurate values for o_{emp} and o_w by using least squares method, but for this small test, we considered the more primitive equation solving above, sufficient.

Last, we can estimate o_{eo} by keeping the assumptions that $t_e = t_x = 0$ and find the average of the values we get by inserting data from 1. run and the estimates for o_{emp} and o_w in equation 1. This gives us

$$o_{\text{eo}} \approx 0.2 \quad (5)$$

which may become larger if more projects has joined DCF, stressing the employment office. All in all we have now got an estimation for the total execution time

$$T \approx 0.2n_w + 0.3n_j + \frac{n_j}{n_w}(1.4 + t_x + t_e) \quad (6)$$

Now, in figure 7 on the following page, we let $t_e = 30$ and test if our hypothesis looks right. As seen in the figure, our two estimations are close to the actual execution time, which indicates that our model is acceptable.

n_w	n_j	1. run	2. run	T_k/T	Our estimate
16	64	156.4	147.0	0.5	147.7
8	64	285.5	285	-	271.8

Figure 7: The tabular shows the average execution times for the 2 tests having $t_e = 30$ seconds and at each row $k = 2$.

6.3.2 Security

Now we look at the security handling. We have created four test runs that each tries to violate a security policy. In appendix C on page 61 we have listed the output from the test runs. As seen, all violations are caught and after each caught violation, the worker disconnects from the employer and reconnects to the employment office in order to be employed at a new employer, which is hopefully more trust worthy.

6.3.3 Real life

For the real life test we have done a number of runs with the parameterizable function. We have chosen a quite high probability of 10% for the worker to deliver a wrong result – just to stress the framework a bit. The function execution time at the employer is set to 30 seconds and the possible worker execution times from 9 to 90 seconds. The tests have been run with 16 workers, 64 data segments and with both a demand of 2 and 3 equal results. The results are shown in figure 8.

Equals to see	Time	Wrong results
2	637	1
2	693	3
2	696	2
2	552	2
2	576	3
2	550	2
3	1060	2
3	1021	0
3	1003	0
3	791	0
3	889	0
3	823	1

Figure 8: Runtimes for our real life test runs. The “Wrong results” column lists the number of wrong results that makes it through the framework and up to the programmer. The average runtime with a requirement of 2 equal results is 617 seconds and 2.2 wrong results received. For a requirement of 3 equal results the averages are 931 seconds and 0.5 wrong results.

As the employment office uses a worker timeout of 2 times the local execution time, all workers with return times in the interval 60-90 seconds (37% of the workers) will timeout and new jobs will therefore be scheduled. For a requirement of two equal results, 128 jobs are initially scheduled. 37% of these (≈ 47) will timeout and will therefore be rescheduled. Again, 37% of the 47 (≈ 17) will timeout and so on. This gives us 198 jobs in total. Of these 198,

10% (≈ 20) will deliver a false result and must therefore be rescheduled – and 10% of the 20 again. All in all, we will in average produce 220 jobs to be calculated. Similarly we will get 334 jobs at the requirement of 3 equals. As the average execution time is 49.5 seconds we should get an average total execution time of 769 seconds with a requirement of 2 equal results and 1166 seconds at a requirement of 3 equal results (using equation 6 on page 34). A real life execution time may, however, vary a lot from this estimate depending on the worker speeds and wrong result probability in the test runs. Also the fact, that we accept timed out result deliveries, could explain that our real life test run execution times is lower than the above two estimates.

As seen in the figure 8 a larger requirement of equal values resulted in a lower error rate but did not totally eliminate the (relatively many) received wrong results. In a real life situation we guess one could expect a much lower rate of wrong results and thus we think a requirement of 2 equal results would be sufficient in most real life projects.

7 Conclusion

As stated in section 1.3 on page 2 we have achieved all the goals we set. As we saw in the test section (section 6 on page 31) the system works in practice and should be able to perform in stressed environments.

We have unfortunately not been able to really test the scalability of the framework since we do not have access to the number of different hosts that this would require but we have tested with up to 16 different hosts and the framework copes with this without problems. We have in our tests observed an overhead of about two seconds per job not counting network transfer (see section 6 on page 31), which in our view is acceptable considering that the normal execution time of the function would most likely be several orders of magnitude larger than this overhead. We are also confident that we have resolved all the security issues which we have identified.

7.1 Possible enhancements for a future version

There are a number of things that could be improved for a future version of DCF. A number of these is listed below.

SSL Currently only plain non encrypted communication is supported. Security would be improved if the DCF also could handle communication over SSL connections.

Compress traffic Depending on the project, the function results may become large. Eg. if the project was a rendering engine which distributed rendering of small movie sequences to workers, the network traffic may become high. If data was compressed before sending it, bandwidth could be saved. There are, however, also the chance for the majority of network packages to be small, eg. if most functions only return booleans. In that case, the compression header could increase the connection load. So the benefit from compressing is determined of the DCF usage pattern.

Worker timeout at employers Currently, workers are never timing them self out and explicitly sends a `Disconnect` package to the employer after being employed. Disconnects are currently a result of the socket timeout that we have set. If there are lack of workers, it could be advantageous to let them time out periodically, e.g. after being employed for a couple of hours or after having performed a certain amount of calculations. The first approach has the drawback that projects with really time consuming functions will only get at most one data segment calculated per employer, this making the benefit of this feature depend on the number of workers subscribed to the DCF network. The second approach could be interesting to implement.

Specs from worker BOINC provides the ability to collect data from the workers and using it for statistics and for a more fine graded delegating of jobs to workers fulfilling specific needs. As we decided to let workers be as anonymous as possible, we have chosen not to implement this feature. But one solution could be to let the workers write their host specifications them self in the configuration file and let the system handle if no host information was given. This approach both gives the benefit of projects being able to chose client configuration (speed, memory amount, etc.) and makes it possible for the worker to hide its host configuration. Such additional usage of worker host information gathering would demand a redesign of the scheduling at the employment office.

Worker statistics Currently, the worker presents only accounting for the number of jobs performed at the current employer and the total number of jobs performed. And these numbers are both cleared when the worker restarts. It could be interesting for the worker user to see more detailed statistics for his/her work performance: Eg. number of different projects being calculated on, total calculation time, etc.

Resource limiting We have no possibility to adjust resource usage at the worker other than the network bandwidth usage. Skilled worker users might be able to start the worker with a low execution priority but that is not a preferable solution. Also memory usage is currently only limited by the Java virtual machine. Another approach could be to let the worker listen for the system activity and only request for employment if the system had stood idle for some period of time. That is, letting the worker act like a kind of screen saver.

Temporary directory As we have currently no support for resource limiting, eg. limiting the total amount of disk space usage, we have chosen to totally forbid disk writing. With a resource handler, we could provide the possibility for functions to create files in a temporary directory.

Cleanup requests The employer `halt()` method should send a cleanup package to the employment office in order to delete its pending requests. Pending requests causes workers to connect to the employer with non valid keys, which is simply waste of time.

Lasting connection threads Instead of spawning a new thread at each incoming connection we could increase scalability if we used a pool of already started threads. This is a well known technique used in e.g. the Apache web server, and we might decrease the DCF overhead and thus benefit from implementing it as well.

Timeout in send We have a serious problem if the set of package receivers choose to receive data really slowly, like eg. 1 byte per minute. If the number of such receivers exceeds the maximal number of simultaneous connections they can totally stall the system. If the send functionality measured a timeout based on the package size we could prevent this for happening.

Use estimator result As a minor improvement we could use the result returned from the local estimator at the employer and thus saving a single calculation per function. Currently the result is just thrown away.

Worker disconnects As for now, worker disconnects (either by an interrupted connection or by an explicit disconnect package) is not handled if the worker lies in a worker pool. At the employer this causes us to waste time when we try to send jobs to a now closed socket. At the employment office the problem is worse, because a dead socket makes us remove a worker request without having forwarded a real worker.

Secure box in employer can be eliminated Our use of the secure box at the employer has shown to be an unnecessary mechanism (despite at the worker): All serialized incoming data is being unserialized based on the code at the employer, so an eventually overwritten and harmful method in the received objects will not be able to survive the network transfer. By eliminating the secure box we could decrease the employer overhead.

Smarter project edit The project editing in the project manager is not optimal. Its usability could be increased by allowing the user to *not* reenter a project title and project description and by easier editing contacts.

References

[SETI] <http://setiathome.ssl.berkeley.edu>

[BOINC] <http://boinc.berkeley.edu>

[MPI] <http://www-unix.mcs.anl.gov/mpi>

[SECURITY] <http://java.sun.com/j2se/1.4.2/docs/guide/security>

[DCFHOME] <http://www.resen.org/~rra/dcf>

[JDBCHOME] <http://jdbc.postgresql.org/download.html>

A Programmers guide

This is the programmers guide to DCF. We remind of the three types of entities of DCF:

The employer That is you.

The employment office A central register at which you must register your project.

The workers This is all the volunteers that share their computing power for your project.

Here, we will explain how to create a project that makes use of the workers subscribed to the DCF network and how to use the DCF API to get the calculations done.

All source code below is Java due to the fact that DCF is only implemented to be used with this language.

A.1 Managing the project

In order to request workers, your employer must have a project ID and a matching key for this ID. These are sent to you when your project is being registered at the employment office. As default, the ID and key for the project and the your contact data is stored and later read from a config file, `employment_info_0.cfg`. However this name can be changed in the employer configuration. Normally a project will only be placed on one host, but if your project has several contacts, their connection data will be written together with the project ID and key at corresponding `.cfg` files.

Managing your project registration can be done in two ways: Through the DCF API or by the `ProjectManager` program. The `ProjectManager` is evoked by typing

```
java employer/ProjectManager
```

in the DCF dir. The `ProjectManager` now guides you through the registration. Notice: You should normally let the `ProjectManager` read employment office IP and port from file.

When your project is added and the `ProjectManager` has created the file `employment_info_0.cfg`, you are ready to rock. An example session can be viewed in figure 9 on the next page.

A.2 The API

A sketch DCF program consist of

1. inclusion of `employer.*` and `employer.interfaces.*`.
2. initalization of DCF
3. creation of function bodies, DCF queues, enqueueing of data segments and dequeuing of data segment results
4. halt DCF

An example program is shown in figure 10 on page 43. The four steps above is done at line 1-2, 7, 10-19 and 22 resp.

Lets look a bit more closely on the example in figure 10. At line 13 an instance of `DataSegment` is created. This is the segment that will be sent to one or more workers.

```
] java employer/ProjectManager
-- Project Manager started --
Employment office info read from employment_office_info.cfg
Employment office set to: dcf.resen.org:1280
Is this correct? (y/n):y
=====
1. Add project
2. Edit project
3. Delete project
4. Exit
=====
Make you selection: 1

-- Add project selected --
Enter title:
My title
Enter description:
Some description should be here
Enter contact 0 as <host:port> (enter 0 for no more cotacts):
ask.diku.dk:1229
Enter contact 1 as <host:port> (enter 0 for no more cotacts):
0
Adding project... DONE
=====
1. Add project
2. Edit project
3. Delete project
4. Exit
=====
Make you selection: 4
```

Figure 9: A typical ProjectManager session

```

1  import employer.*;
2  import employer.interfaces.*;
3
4  public class MyClass {
5      public static void main(String [] args) {
6          try {
7              DCF.init (); // Init the project
8
9              // Do work start -----
10             Function f = new MyFunction ();
11             Queue Q = DCF.getDisQ (f, "MyClassDefinitions.jar");
12
13             DataSegment data = new MyDataSegment(<some values>);
14             long sequence = Q.push(data);
15
16             QSegmentResult qResult = Q.get ();
17             MyDataSegmentResult realResult =
18                 (MyDataSegmentResult) qResult.getDataSegmentResult ();
19             System.out.println ("Result:_ " + realResult);
20             // Do work end -----
21
22             DCF.halt (); // Halt DCF
23         }
24         catch (Exception e) {
25             System.out.println ("An_error_occured:_ " + e);
26         }
27     }
28 }

```

Figure 10: A very simple and almost minimal DCF program that pushes one data segment to a discontinuous queue, gets the result and prints it.

`DataSegments` must implement the interface shown in figure 11. At line 10, an instance of `MyFunction` is made. As the `DataSegment`, the function must implement the interface shown in figure 12. At line 11, we create a discontinuous execution queue, that will make enqueued `DataSegments` be evaluated by the function. Note, that a path to a `.jar`-file containing the class definitions relevant for executing the code, must be given as argument. We push the data at line 14 and receives its corresponding `DataSegmentResult` again at line 16-18. `DataSegmentResults` must implement the interface shown in figure 13. This is basically the idea of DCF.

```

1 package employer.interfaces;
2
3 public interface DataSegment extends java.io.Serializable {}

```

Figure 11: The `DataSegment` interface.

```

1 package employer.interfaces;
2
3 public interface Function extends java.io.Serializable {
4
5     public DataSegmentResult doCalculation(DataSegment d);
6 }

```

Figure 12: The `Function` interface. The `doCalculation()` method performs the calculation.

```

1 package employer.interfaces;
2
3 public interface DataSegmentResult extends java.io.Serializable{
4
5     public boolean isEqualTo(DataSegmentResult r);
6 }

```

Figure 13: The `DataSegmentResult` interface. The `isEqualTo()` method is used to compare two `DataSegmentResults`

Below we are describing the API in details.

A.2.1 DCF

```
void static init()
    Inits the employer
```

```
void static halt()
    Stops the employer
```

```
Queue static getDisQ(DCFFunction, String)
    Returns a discontinuous execution queue. The DCFFuntion will be performed on en-
```

queued data segments. The `String` is a path to a `.jar`-file, containing the class definitions relevant for making instances of `DCFFunction`, `DataSegment` and `DataSegmentResult`. Results becomes available at discontinuous as soon as they have been completely calculated.

`Queue static getConQ(DCFFunction, String)`

As `getDisQ()`, returns a continuous queue. This works as the discontinuous execution queue, except that results will be available for dequeuing in the same order as they were enqueued.

`static void setBacklog(int)`

Sets the backlog size.

`static void setBindInetAddress(InetAddress)`

The local IP that the employer should listen to. Default is `null` (all IP addresses of this machine).

`static void setBindPort(int)`

The local port that the employer runs on. Default is 1229.

`static void setContactId(long)`

This set the local contact ID. Default is read from the `employer_info_0.cfg` file.

`static void setEmploymentOfficeIP(InetAddress)`

Set host address for employment office. Default is read from the `employer_info_0.cfg` file.

`static void setEmploymentOfficePort(int)`

Set port for employment office. Default is read from the `employer_info_0.cfg` file.

`static void setEqualsToSee(int)`

This sets the number of equal results that the employer must see, before delivering the result to the `Queue` for dequeuing. Default is 2. Notice that a higher value increases the security for getting right results but also increases runtime as each data segment has to be calculated more times.

`static void setFilePath(String)`

The top level file path. All files (logs and config files) will be placed relatively to this path.

`static void setLocalKeySize(int)`

The length of keys generated locally. These are send to workers when they are send the function body and data segment. The key must be valid when they return the result. Default is 12.

`static void setLoggingFilename(String)`

Filename on logfile.

`static void setLoggingLevel(Level)`

The logging level. Default is `INFO`.

`static void setMaxBandwidthPerReceive(double)`

This is the maximum bandwidth used in each receive in KB/s. In combination with `setMaxSimulSends()` it can be used to control how much bandwidth is used at any time. A rate of 0 means no limit. Default is 0 (no limit).

`static void setMaxBandwidthPerSend(double)`

As `setMaxBandwidthPerReceive()`, except that this method adjust max bandwidth per send.

`static void setMaxPackageSize(int)`

This is the max size in byte of any package that you will receive. Any packages longer than this will be discarded. The motivation for this option is to limit the amount of garbage that one renegade worker can send. Default is 10 MB.

`static void setMaxSimulSends(int)`

This is the maximum of simultaneous sends that can occur at any moment. Sends here refer to sends of jobs to workers. In combination with the `maxBandwidth()` per send, it can be used to control how much bandwidth is used max. Default is 10.

`static void setProjectId(long)`

Sets the project ID. Default is read from the `employer_info_0.cfg` file.

`static void setProjectKey(byte[])`

Sets the project key. Default is read from the `employer_info_0.cfg` file.

`static void setWorkerPoolSize(int)`

The amount of inactive workers that you can have assigned at any one time. Notice that this also affects how many open connections are maintained at the same time, since every worker is associated with a TCP connection. Default is 100.

A.2.2 Queue

`public long push(DataSegment d)`

Returns sequence number of segment pushed.

`public long push(DataSegment[] d)`

Returns sequence number of last segment pushed.

`public QSegmentResult get()`

Blocks until the next result is ready.

`public QSegmentResult[] get(int n)`

Blocks until next *n* results are ready.

`public QStats getQStats()`

Returns a `QStats` object containing various statistics for the queue.

A.2.3 Configuration file

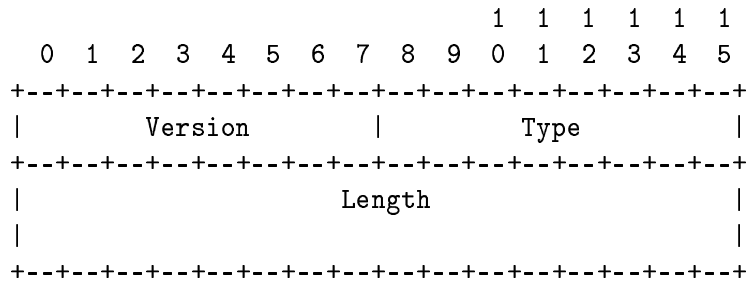
The employers configuration properties can be read from a configuration file named `employer.cfg`. If the file exists, DCF reads all relevant properties at `init()`. `employer.cfg` should have the format

```
Name1 = value1
Name2 = value2
...
```

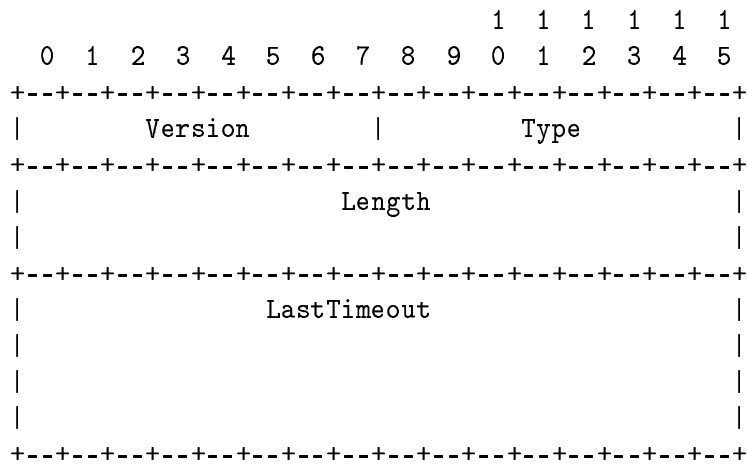
where all `Namei` can be the property names described in section A.2.1 on page 44. E.g., the function `DCF.setEqualsToSee` represents the property `EqualsToSee`.

B Package layouts

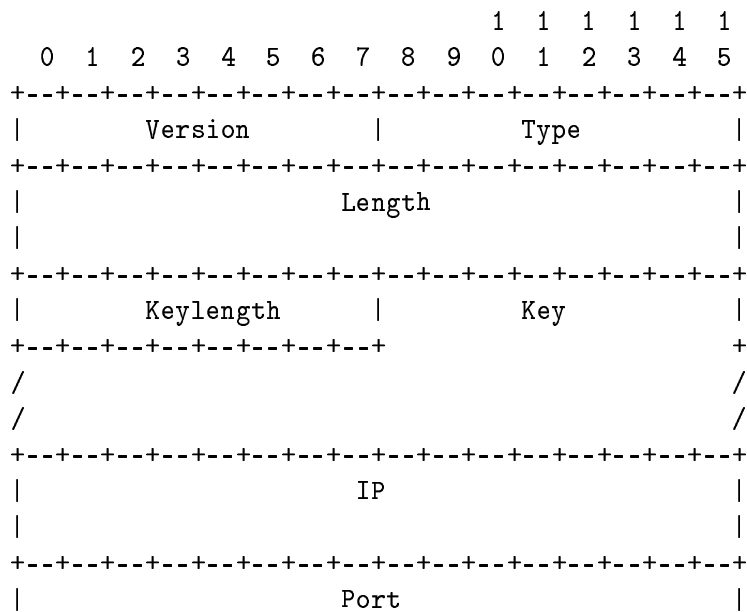
B.1 Header



B.2 WorkReq (type 15)



B.3 ConToEmp (type 17)



```

|
+-----+

```

B.4 Reconnect (type 16)

```

                                1 1 1 1 1 1
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
+-----+-----+-----+-----+-----+
|          Version          |          Type          |
+-----+-----+-----+-----+-----+
|                          |          Length          |
|                          |                          |
+-----+-----+-----+-----+-----+
|          Action          |          0          |
+-----+-----+-----+-----+-----+
|                          |          Timeout          |
|                          |                          |
|                          |                          |
|                          |                          |
+-----+-----+-----+-----+-----+
|                          |          IP          |
|                          |                          |
+-----+-----+-----+-----+-----+
|                          |          Port          |
|                          |                          |
+-----+-----+-----+-----+-----+

```

B.5 EmpReq (type 12)

```

                                1 1 1 1 1 1
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
+-----+-----+-----+-----+-----+
|          Version          |          Type          |
+-----+-----+-----+-----+-----+
|                          |          Length          |
|                          |                          |
+-----+-----+-----+-----+-----+
|          Keylength          |          Key          |
+-----+-----+-----+-----+-----+
/                                                                    /
/                                                                    /
+-----+-----+-----+-----+-----+

```

B.6 EmpReqAck (type 2)

```

                                1 1 1 1 1 1
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
+-----+-----+-----+-----+-----+

```

```

|          Version          |          Type          |
+-----+-----+-----+-----+-----+-----+
|                          |                        |
|                          |                        |
+-----+-----+-----+-----+-----+-----+
|                          |                        |
|                          |                        |
|                          |                        |
|                          |                        |
+-----+-----+-----+-----+-----+

```

B.7 EmpReqDen (type 1)

```

                                1 1 1 1 1 1
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
+-----+-----+-----+-----+-----+-----+
|          Version          |          Type          |
+-----+-----+-----+-----+-----+-----+
|                          |                        |
|                          |                        |
+-----+-----+-----+-----+-----+-----+
|          Reason          |                        |
+-----+-----+-----+-----+-----+

```

B.8 PerformJob (type 4)

```

                                1 1 1 1 1 1
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
+-----+-----+-----+-----+-----+-----+
|          Version          |          Type          |
+-----+-----+-----+-----+-----+-----+
|                          |                        |
|                          |                        |
+-----+-----+-----+-----+-----+-----+
|          Keylength       |          Key           |
+-----+-----+-----+-----+-----+-----+
/                                                                    /
/                                                                    /
+-----+-----+-----+-----+-----+-----+
|                          |                        |
|                          |                        |
|                          |                        |
|                          |                        |
+-----+-----+-----+-----+-----+-----+
|                          |                        |
|                          |                        |
|                          |                        |
|                          |                        |
+-----+-----+-----+-----+-----+-----+
|          Jarfile length  |                        |
|                          |                        |
+-----+-----+-----+-----+-----+-----+

```

```

/                               Jarfile                               |
/                               |                                     |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                               Function+datalength                   |
|                               |                                     |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
/                               Function                               /
/                               |                                     /
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
/                               Datasegment                           /
/                               |                                     /
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

```

B.9 Disconnect (type 3)

```

                                1 1 1 1 1 1
  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|           Version           |           Type           |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                               Length                   |
|                               |                         |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|           Reason           |                         |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

```

B.10 DeliverResult (type 14)

```

                                1 1 1 1 1 1
  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|           Version           |           Type           |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                               Length                   |
|                               |                         |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|           Keylength         |           Key           |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
/                               |                         /
/                               |                         /
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                               WID                       |
|                               |                         |
|                               |                         |
|                               |                         |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                               JID                       |
|                               |                         |

```

```

|
|
|
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                               Resultlength                               |
|
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
/                               Result                                   /
/
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

```

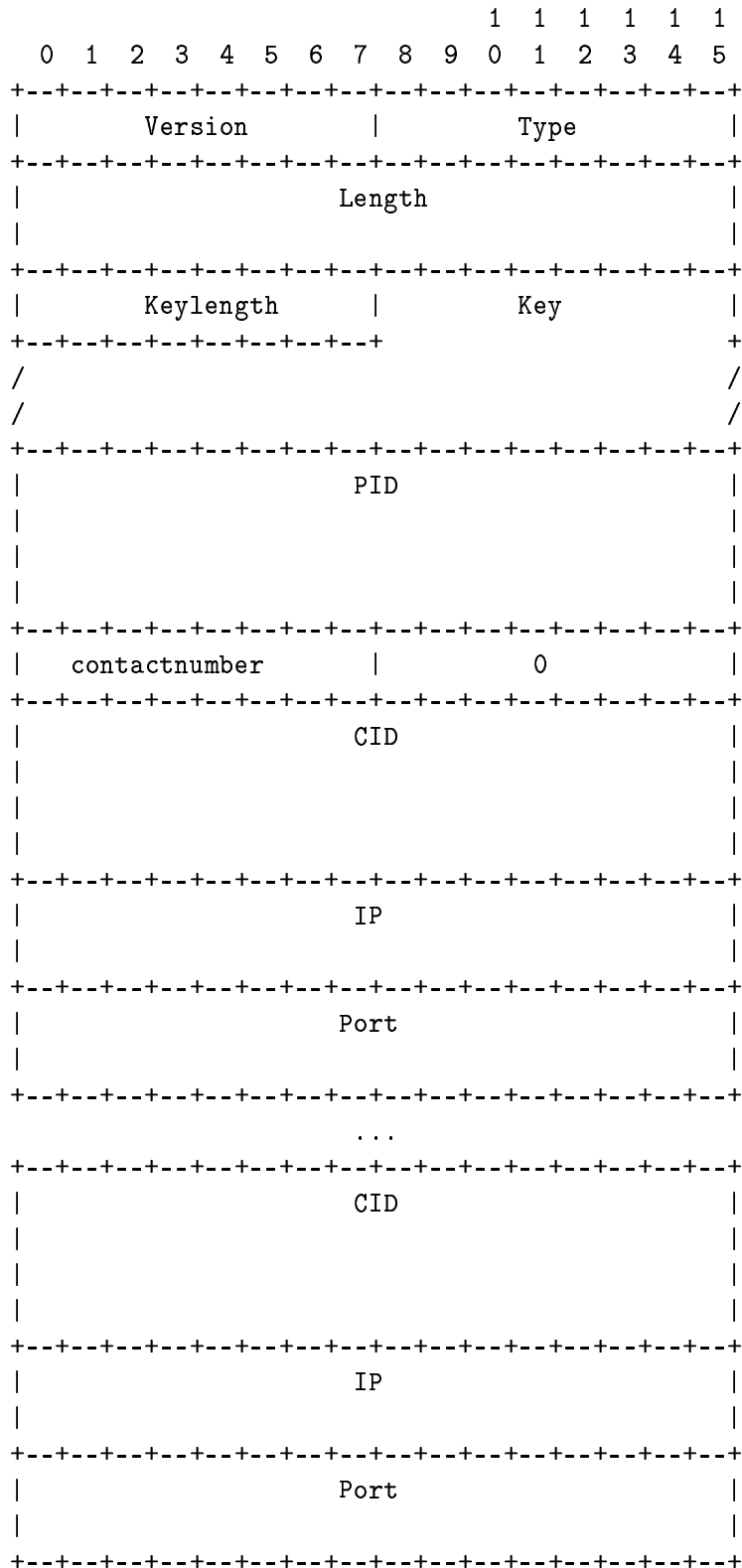
B.11 AddProject (type 8)

```

                                1 1 1 1 1 1
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|      Version      |      Type      |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                               Length                               |
|
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|      Titlelength  |      Descriptionlength  |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
/                               Title                                   /
/
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
/                               Description                               /
/
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|      Contactnumber  |      0      |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                               IP                               |
|
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                               Port                               |
|
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                               ...                               |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                               IP                               |
|
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|                               Port                               |
|
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

```

B.12 AddProjectAck (type 18)

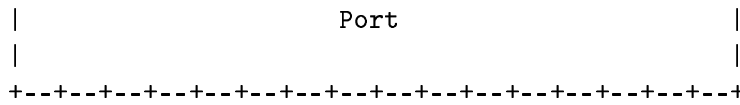


B.13 AddProjectDen (type 19)

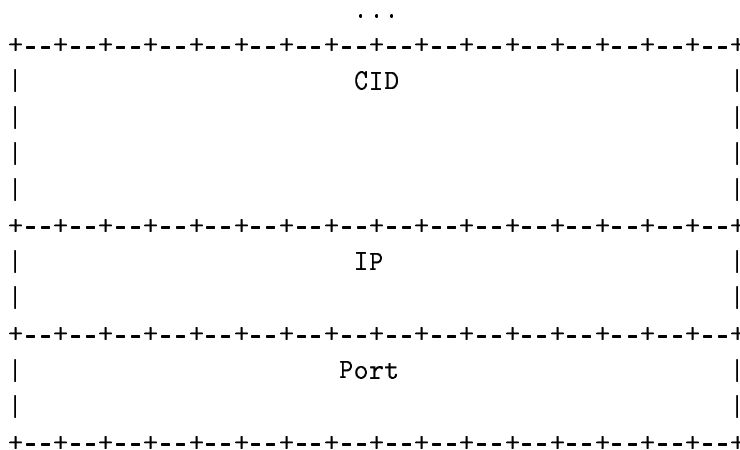
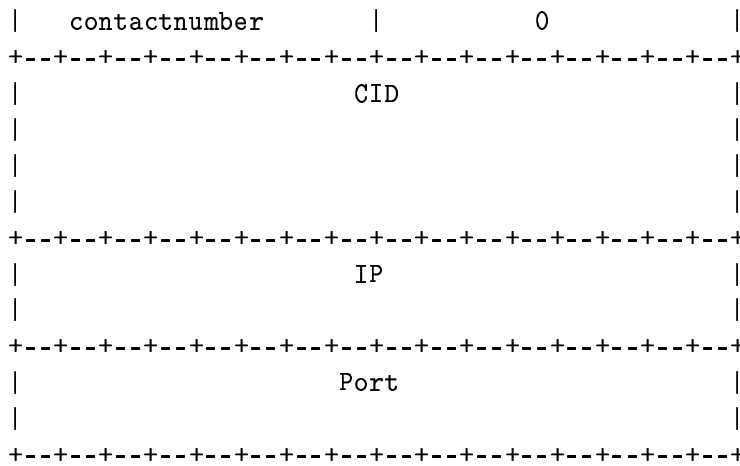
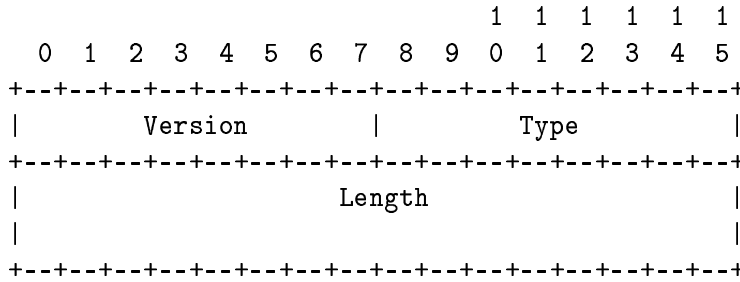
											1	1	1	1	1	1
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	
Version									Type							
Length																
Reason																

B.14 EditProject (type 9)

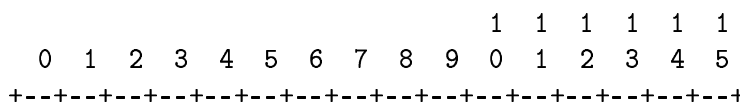
											1	1	1	1	1	1
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	
Version									Type							
Length																
Keylength									Key							
/ /																
PID																
Titlelength									Descriptionlength							
/ Title /																
/ Description /																
Editnumber									0							
CID																



B.15 EditProjectAck (type 20)



B.16 EditProjectDen (type 21)



```

|      Version      |      Type      |
+-----+-----+
|                      Length                      |
|
+-----+-----+
|      Reason      |
+-----+-----+

```

B.17 DelProject (type 10)

```

                                1 1 1 1 1 1
      0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
+-----+-----+-----+-----+-----+-----+
|      Version      |      Type      |
+-----+-----+-----+-----+-----+-----+
|                      Length                      |
|
+-----+-----+-----+-----+-----+-----+
|      Keylength    |      Key      |
+-----+-----+-----+-----+-----+-----+
/
/
+-----+-----+-----+-----+-----+-----+
|                      PID                      |
|
|
|
|
+-----+-----+-----+-----+-----+-----+

```

B.18 DelProjectAck (type 22)

```

                                1 1 1 1 1 1
      0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
+-----+-----+-----+-----+-----+-----+
|      Version      |      Type      |
+-----+-----+-----+-----+-----+-----+
|                      Length                      |
|
+-----+-----+-----+-----+-----+-----+

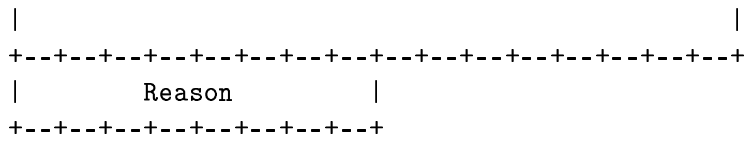
```

B.19 DelProjectDen (type 23)

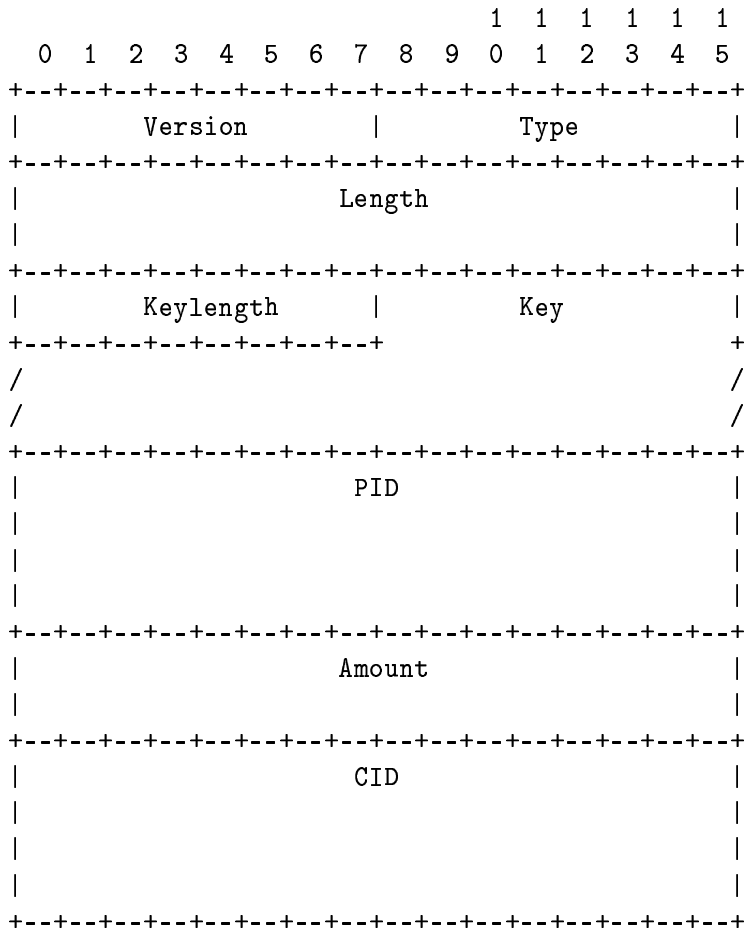
```

                                1 1 1 1 1 1
      0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
+-----+-----+-----+-----+-----+-----+
|      Version      |      Type      |
+-----+-----+-----+-----+-----+-----+
|                      Length                      |
|

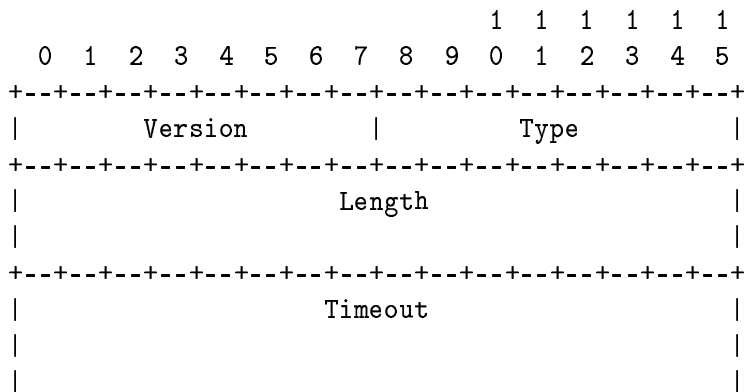
```



B.20 WorkersReq (type 5)



B.21 WorkersReqAck (type 6)



```

|
+-----+-----+-----+-----+-----+-----+-----+-----+
|                                     Requests                                     |
|
+-----+-----+-----+-----+-----+-----+-----+-----+
|      Keylength      |      Keynumber      |
+-----+-----+-----+-----+-----+-----+-----+-----+
/                                     Keys                                     /
/                                     /
+-----+-----+-----+-----+-----+-----+-----+-----+

```

B.22 WorkersReqDen (type 7)

```

                                1 1 1 1 1 1
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
+-----+-----+-----+-----+-----+-----+-----+-----+
|      Version      |      Type      |
+-----+-----+-----+-----+-----+-----+-----+-----+
|                                     Length                                     |
|
+-----+-----+-----+-----+-----+-----+-----+-----+
|      Reason      |
+-----+-----+-----+-----+-----+-----+-----+-----+

```

B.23 SyncReq (type 11)

```

                                1 1 1 1 1 1
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
+-----+-----+-----+-----+-----+-----+-----+-----+
|      Version      |      Type      |
+-----+-----+-----+-----+-----+-----+-----+-----+
|                                     Length                                     |
|
+-----+-----+-----+-----+-----+-----+-----+-----+
|      Keylength    |      Key      |
+-----+-----+-----+-----+-----+-----+-----+-----+
/
/
+-----+-----+-----+-----+-----+-----+-----+-----+
|                                     PID                                     |
|
|
|
+-----+-----+-----+-----+-----+-----+-----+-----+
|                                     CID                                     |
|
|

```

```

|
+-----+

```

B.24 SyncReqAck (type 24)

```

                                1 1 1 1 1 1
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
+-----+-----+-----+-----+-----+
|          Version          |          Type          |
+-----+-----+-----+-----+-----+
|                          |          Length          |
|                          |                          |
+-----+-----+-----+-----+-----+
|                          |          Requests          |
|                          |                          |
+-----+-----+-----+-----+-----+

```

B.25 SyncReqDen (type 25)

```

                                1 1 1 1 1 1
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
+-----+-----+-----+-----+-----+
|          Version          |          Type          |
+-----+-----+-----+-----+-----+
|                          |          Length          |
|                          |                          |
+-----+-----+-----+-----+-----+
|          Reason          |                          |
+-----+-----+-----+-----+-----+

```

C Security test results

The output from our four security test runs are shown below. As all four security violations were caught, we are satisfied with the security handling.

C.1 File read

This test tries to read the `dcf.log` file.

```
java worker/Worker -v
Reading configuration from worker.cfg
Establishing connenction with employment office (ask.diku.dk/130.225.96.225:1280)
Connection Established
Applying for work
Employment office processing work application
Responce received from employment office
Processing responce
Connect to employer received
Connecting to new employer (/130.225.96.225:1229)--
Connection established
Requesting employment
Wating for responce
Responce received
Yes! I got employed!
Now waiting for work
Package received from employer
Workunit received
Creating temp and classes dir
Creating a secure box to process function
Passing key to secure box
Passing function and data to secure box
Performing work
Retreiving result from the secure box (yes, Im beeing careful)
SECURE BOX ERROR BEGIN
java.security.AccessControlException: access denied (java.io.FilePermission dcf.log read)
    at java.security.AccessControlContext.checkPermission(AccessControlContext.java:270)
    at java.security.AccessController.checkPermission(AccessController.java:401)
    at java.lang.SecurityManager.checkPermission(SecurityManager.java:542)
    at java.lang.SecurityManager.checkRead(SecurityManager.java:887)
    at java.io.FileInputStream.<init>(FileInputStream.java:100)
    at employer.tests.SecurityFunction.doCalculation(TestSecurity.java:89)
    at worker.SecureBox.main(SecureBox.java:47)
SECURE BOX ERROR END
An IO error occured
Establishing connenction with employment office (ask.diku.dk/130.225.96.225:1280)
```

C.2 File write

This test tries to create a file named `SecurityTestFile` in the `./` directory.

```
java worker/Worker -v
Reading configuration from worker.cfg
Establishing connenction with employment office (ask.diku.dk/130.225.96.225:1280)
Connection Established
Applying for work
Employment office processing work application
Responce received from employment office
```

```

Processing responce
Connect to employer received
Connecting to new employer (/130.225.96.225:1229)--
Connection established
Requesting employment
Wating for responce
Responce received
Yes! I got employed!
Now waiting for work
Package received from employer
Workunit received
Creating temp and classes dir
Creating a secure box to process function
Passing key to secure box
Passing function and data to secure box
Performing work
Retreiving result from the secure box (yes, Im beeing careful)
SECURE BOX ERROR BEGIN
java.security.AccessControlException: access denied (java.io.FilePermission SecurityTestFile write)
    at java.security.AccessControlContext.checkPermission(AccessControlContext.java:270)
    at java.security.AccessController.checkPermission(AccessController.java:401)
    at java.lang.SecurityManager.checkPermission(SecurityManager.java:542)
    at java.lang.SecurityManager.checkWrite(SecurityManager.java:975)
    at java.io.File.createNewFile(File.java:824)
    at employer.tests.SecurityFunction.doCalculation(TestSecurity.java:98)
    at worker.SecureBox.main(SecureBox.java:47)
SECURE BOX ERROR END
An IO error occured
Establishing connenction with employment office (ask.diku.dk/130.225.96.225:1280)

```

C.3 Socket access

This test tries to connect to the URL <http://www.google.com>.

```

java worker/Worker -v
Reading configuration from worker.cfg
Establishing connenction with employment office (ask.diku.dk/130.225.96.225:1280)
Connection Established
Applying for work
Employment office processing work application
Responce received from employment office
Processing responce
Connect to employer received
Connecting to new employer (/130.225.96.225:1229)--
Connection established
Requesting employment
Wating for responce
Responce received
Yes! I got employed!
Now waiting for work
Package received from employer
Workunit received
Creating temp and classes dir
Creating a secure box to process function
Passing key to secure box
Passing function and data to secure box
Performing work
Retreiving result from the secure box (yes, Im beeing careful)

```

SECURE BOX ERROR BEGIN

```
java.security.AccessControlException: access denied (java.net.SocketPermission www.google.com resolve)
  at java.security.AccessControlContext.checkPermission(AccessControlContext.java:270)
  at java.security.AccessController.checkPermission(AccessController.java:401)
  at java.lang.SecurityManager.checkPermission(SecurityManager.java:542)
  at java.lang.SecurityManager.checkConnect(SecurityManager.java:1042)
  at java.net.InetAddress.getAllByName0(InetAddress.java:909)
  at java.net.InetAddress.getAllByName0(InetAddress.java:890)
  at java.net.InetAddress.getAllByName(InetAddress.java:884)
  at java.net.InetAddress.getByName(InetAddress.java:814)
  at sun.net.www.http.HttpClient.<init>(HttpClient.java:282)
  at sun.net.www.http.HttpClient.<init>(HttpClient.java:253)
  at sun.net.www.http.HttpClient.New(HttpClient.java:321)
  at sun.net.www.http.HttpClient.New(HttpClient.java:306)
  at sun.net.www.http.HttpClient.New(HttpClient.java:301)
  at sun.net.www.protocol.http.HttpURLConnection.plainConnect(HttpURLConnection.java:463)
  at sun.net.www.protocol.http.HttpURLConnection.connect(HttpURLConnection.java:454)
  at sun.net.www.protocol.http.HttpURLConnection.getInputStream(HttpURLConnection.java:556)
  at java.net.URLConnection.getContent(URLConnection.java:582)
  at java.net.URL.getContent(URL.java:969)
  at employer.tests.SecurityFunction.doCalculation(TestSecurity.java:102)
  at worker.SecureBox.main(SecureBox.java:47)
```

SECURE BOX ERROR END

An IO error occurred

Establishing connection with employment office (ask.diku.dk/130.225.96.225:1280)

C.4 Start process

This test tries execute the following shell command: touch /tmp/SecurityTestFile.

```
java worker/Worker -v
Reading configuration from worker.cfg
Establishing connection with employment office (ask.diku.dk/130.225.96.225:1280)
Connection Established
Applying for work
Employment office processing work application
Response received from employment office
Processing response
Connect to employer received
Connecting to new employer (/130.225.96.225:1229)--
Connection established
Requesting employment
Waiting for response
Response received
Yes! I got employed!
Now waiting for work
Package received from employer
Workunit received
Creating temp and classes dir
Creating a secure box to process function
Passing key to secure box
Passing function and data to secure box
Performing work
Retrieving result from the secure box (yes, Im beeing careful)
SECURE BOX ERROR BEGIN
java.security.AccessControlException: access denied (java.io.FilePermission <<ALL FILES>> execute)
  at java.security.AccessControlContext.checkPermission(AccessControlContext.java:270)
  at java.security.AccessController.checkPermission(AccessController.java:401)
```

```
at java.lang.SecurityManager.checkPermission(SecurityManager.java:542)
at java.lang.SecurityManager.checkExec(SecurityManager.java:799)
at java.lang.Runtime.exec(Runtime.java:547)
at java.lang.Runtime.exec(Runtime.java:416)
at java.lang.Runtime.exec(Runtime.java:358)
at java.lang.Runtime.exec(Runtime.java:322)
at employer.tests.SecurityFunction.doCalculation(TestSecurity.java:106)
at worker.SecureBox.main(SecureBox.java:47)
SECURE BOX ERROR END
An IO error occurred
Establishing connection with employment office (ask.diku.dk/130.225.96.225:1280)
```

D Sourcecode

In this section we will describe the directory structure of the source code. The source directory is divided into the following subdirectories:

```
common/  
common/handlers/  
common/implementations/  
common/interfaces/  
common/packages/  
employer/  
employer/handlers/  
employer/implementations/  
employer/implementations/datalayer/  
employer/implementations/joblayer/  
employer/implementations/worklayer/  
employer/interfaces/  
employer/interfaces/datalayer/  
employer/interfaces/joblayer/  
employer/interfaces/worklayer/  
employer/tests/  
employment_office/  
employment_office/handlers/  
employment_office/implementations/  
employment_office/interfaces/  
employment_office/sql/  
employment_office/tests/  
worker/
```

The `employer/`, `employment_office/` and `worker/` directories contains the implementations of these entities and the `common/` directory contains files which are common for all the entities (such as packages). Each of these four directories has an `interfaces/` and `implementations/` subdirectory reflecting the java interface specification and the classes implementing these interfaces. The `handlers/` directories contains connectionhandlers which are used in the `ConnectionManager`. All tests reside in the `tests` directories.

Notice, that we uses the JDBC from postgresql⁷ found at [JDBC_HOME] in the employment office implementation.

The source code is available online at [DCF_HOME] along with documentation generated by the `javadoc` tool. The package layout in this documentation mirrors the layout in the directories shown above.

⁷ pg74.213.jdbc3.jar