DYNAMIC REDUNDANCY SCALING FOR CLIENT LOAD DYNAMICS

Master of Computer Science
Universitetet i Stavanger
The Department of Electrical and Computer Engineering

By
Yinjun Wu
June 2010

Abstract

This paper presents the mechanism and implementation of dynamically redundancy scaling for client load dynamics. The scaling is about the control of maximum and minimum counter of replicas which provide identical service in a distributed computer networks. This mechanism is managed under a Distributed Autonomous Replication Management (DARM) framework which is of autonomous fault treatment supporting and builds upon Spread group communication system.

DARM, like the other existing fault tolerance frameworks, has good solution on fault detecting and toleration aspects; apart from that, DARM is set to always keep recovering a faulty service in a active manner and the recovery is done by generating new replica of the same service in another machine which locates within the same network. The objective of DARM, which is focus more on the improvement of dependability of the system, makes DARM novel among all similar frameworks.

The objective of this dynamical redundancy scaling mechanism built on DARM is to actively and efficiently control the maximum and minimum counter of the replicas in an appropriate state. The maximum and minimum counter will directly effect on the identical-service-providing replicas within a distributed network. When a host is observed with higher CPU load value than a preset threshold, new replicas will be automatically created; similarly, when a host is viewed with lower CPU load, it will be considered to kill if it has not achieved the least replica number that required among the whole network.

This scheme allows system vary the replicas number according to the situation of the service loading all over the network: A new replica can be added to share the burden of whole; an existing replica can be removed to avoid unnecessary resource consuming. The advantages of this schema are: (i) comparing with static control, a dynamic one is more practice-oriented; it has better flexibility for the realistic service loading in a network; (ii) possibility of host crashed due to high loading on top of it is reduced, the principle behind this mechanism is to always balance the existing loading over all active hosts. The approach has been evaluated as an efficient mechanism support to the current DARM framework.

A C implementation of this dynamic control mechanism has been accomplished and introduced in this paper, as well as its testing evaluation. The performance is positive and effective.

Acknowledgments

First and foremost, I would like to thank my supervisor, Hein Meling, the associate professor of the Department of Electrical Engineering and Computer Science at the University of Stavanger who provided me with useful and valuable guidance along with every stage of my thesis implementing period, as well as the paper work. His idea and guidance broaden my understanding of distributed systems which means a lot to me.

I shall extend my thanks to Joakim L.Gilje, the author of DARM, as well for his kindness and help on my thesis. He made very clear explanation on the key functions in DARM which is necessary for me to use in my thesis and he also provided very useful error analyze during the work.

Last but not least, I'd like to thank Theodor Ivesdal, the administrator of Unix Lab in the university, who offered a very good environment for me to write, compile and test my thesis work; Also, I would like to thank my fellow students who supported me a lot during the thesis working.

Contents

Chapter 1

Introduction

Distributed computer networks has been more and more involved in people's daily life, especially on the situation of providing distributed network services, such as bank account management system, email delivery system, etc.. However, a sudden disconnection between nodes in a distributed network is a latent defect, due to the raising reliance on providing services upon distributed networks, developing improved dependability of systems becomes a crucial goal.

A normal way to improve such dependability is to spread multiple replicas of the same service within a geographical area. Those replicas repeat exactly the same functions and are tightly connected to each other through a network while each of them could present failure independently. The Distributed Autonomous Replication Management (DARM) framework is such a self-governed fault treatment framework of improved dependability characteristics. Unlike other existing traditional fault tolerance frameworks, which a centralized system administrator is present to allocate replicas to the failed ones, DARM is achieved to self-managing localizing failures and reconfigurations as its mechanism. Apart from that, a unique and fresh architecture is performed in DARM that not only replica failures are detectable and tolerable, but a faulty service is always recoverable by generating new replicas in some other hosts locate in the same distributed network; And this could be always executed as long as there are available hosts to set substitutable replicas onto which differs DARM from other normal and simple solutions that certain amount of available replicas are needed and hopefully an administrator can managed to replace substitution for those failed replicas before all of them are failed.

DARM was initially inspirited by the idea and pre-work [8, 9, 7] of Hein Meling and later carried on by Joakim in his master thesis [6]. The framework used in this paper is the latest version which is implemented base on the Spread group communication system (GCS) [5], and applies as distributed approach for service replicas placement.

This paper will present a dynamic redundancy scaling for client load dynamics in DARM aimed at providing a more efficient and flexible use of resources in a distributed network through the DARM framework. Consequently, this mechanism improves the dynamical-adjustment ability and self-adaptability of system, where the flexibility of recourses usage and reuse perform better than a static redundancy configuration, and it prevent a human adjustment along with the system running. Dynamic redundancy scaling in DARM is achieved by setting up upper and lower threshold and accumulating or reducing the maximum and minimum replicas in the system. Thereby, the living service replicas can be regulated to the most befitting situation to the actual state automatically and immediately.

What has been implemented in this paper is a mechanism which controls the changing of the existing replicas for an identical service in a distributed network under DARM framework. As it is known that a common way to keep network service dependable is to add replicas of this service and distributed them geographically, and DARM is of such feature; however, when locating those replicas over the network, it comes with other problems, like as how to choose the optimal host machine to locate a new replica and how to control the existing replicas over this network etc.. In order to achieve a better performance on controlling the identical replicas over the network, a dynamic redundancy scaling mechanism is implement under DARM and the principle behind this mechanism is to periodically checking the current loading state over all host machines and make adjustment on replicas when situation comes to different cases, where the gain in terms of improved efficiency and flexibility of recourse utilization in the system.

This dynamic redundancy scaling mechanism differs from its previous version, static redundancy scaling, in that the maximum and minimum replica number will no longer be a fixed value but a variable one which means it will actively and periodically checking the CPU loading of each machines locates in the network and adjust the allowed number for replicas base on observations. This completely control the replica number setting according to the actual loading redundancy and bring a more efficient and flexible utilization of resources in DARM.

1.1 DARM

An autonomous Replication Management (ARM), built by Hein Meling in his previous work [8, 9], is the early form of DARM. The new architecture in DARM has a prototype which extends the Jgroup [11] object group system and implemented base on the Spread group communication system (GCS) [5]. Compare with other similar fault treatment frameworks, DARM reduced the need of a centralized replication manager which controls the global information for all groups while the others required one. Furthermore, DARM's utilization of policy-based management that supports recovery focus on toleration network partitions differ DARM from other frameworks, as well as providing the feature of self-healing and self-configuration in DARM.

1.2 This paper

This paper will begin with an introduction and background to the framework DARM and the dynamic redundancy scaling schema which explain an essential knowledge. A detailed explanation of the implementation will be given and an experiment result will be performed and evaluated after the detailed implementation description. Finally, conclusion will be drawn base on the work and suggestion to future work will be discussed after conclusion.

Chapter 2

Background

In this chapter, some essential knowledge will be presented. First, it will introduce some terms that tightly relevant to the work in DARM; Then, an explanation of DARM will tell something about the communication toolkit DARM relies on, the mechanism inside of DARM and so on; finally, a background of the dynamic redundancy scaling, together with the comparison of the static configuration, will be discussed.

2.1 Term explanation

Some terms that relevant to the work are explained as follow:

1) Distributed network Distributed network is a distributed computing structure where its network resources, like as switching facilities, programmes, processors and data, etc., are spread out through a geographical area and locates on more than one computer.

It prioritizes to the low-cost computer power on the desktop [3], applications and date operations of distributed network are more efficient than local a lot of servers such as area network servers, web servers, regional servers and so forth.

A client/server computing is a popular trend in the distributed network/networking which refers to a view that a user can get a certain capabilities from a client computer while this client computer can request others from other computers that provide services for the clients. [1]

2) Fault tolerant Fault tolerance, or graceful degradation as it sometimes called, is the property that designed to enable a computer-based system or a component to continue a proper operating by a sort of 'back-up' component or programme

when a failure happens to some of it so that it won't be breakdown in these cases. It is not only a property of individual machines but delineate the rules by which they interact as well.

A fault tolerance could be achieved by software, embedded hardware or a combination of those two and especially chased by high-availability or life-critical systems. The decrease is propotional to the severity of the failure if a fault-tolerance operation quality decreases at all, as compared to naively-designed systems which even a small failure will bring a total breakdown. [2]

Usually, there are some essential characteristics that fault tolerance requires which are: a) no single point to repair; b) isolated faults from failed component; c) restrain fault from propagated failure; d) availability of reversion. Besides, fault tolerant systems are featured in either planned or unplanned service outages aspects.

Two keywords, 'replication' and 'redundancy', are addressed as the fundamental characteristics of fault-tolerance by replication. Replication refers providing several identical instance of the same system, leading requests to all of them in parallel and picks the right result base on a quorum; and redundancy represents providing several identical instances of the same system and exchanging to one of the remaining instances when there is a failure. [2]

2.2 DARM

The Distributed Autonomous Replication Management (DARM) framework is a self-governed fault treatment framework which is of the following features: 1) Self-manage localizing failures and reconfiguration. 2) Able to detect and tolerate replica failures. 3) As long as there are available hosts to set substituted replicas, a faulty service will be recovered by generating new replicas in other host sets in the same network.

It is an open source framework and supported on top of Spread group communication system. DARM focuses on deployment and operational aspects aimed at an improved dependability of system through a fault treatment mechanism.

2.2.1 Spread Group Communication Toolkit

As mentioned above that DARM is built on the idea of group communication which was realized on developing on top of Spread GCS which provide group

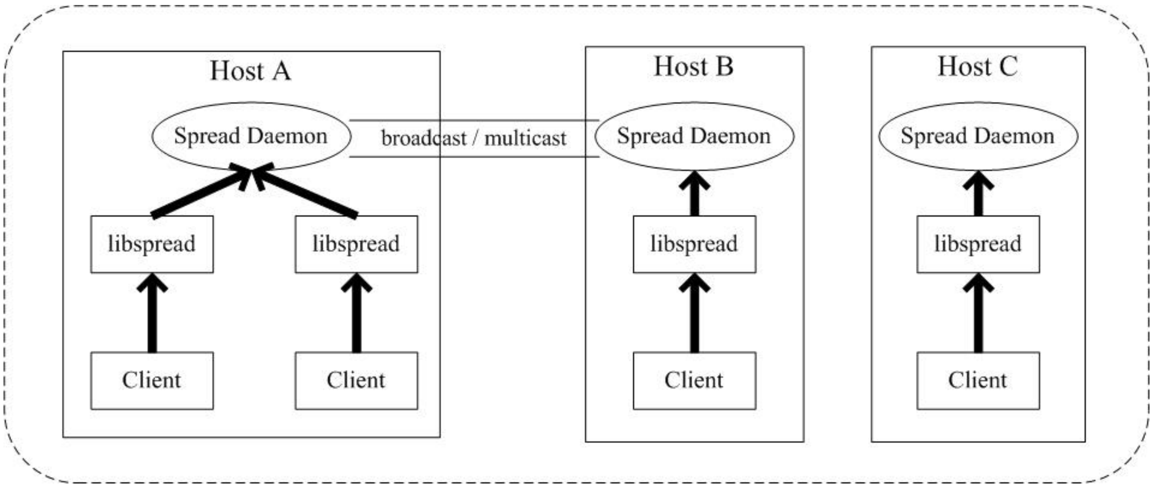


Figure 2.1: Three host with Spread Daemon running in a single LAN

membership services to DARM. Spread is an open source toolkit that provides high performance messaging service that is resilient to faults across local and wide area networks. [4] It enables one or more clients that shared the common interests join into a group aimed at communication and be responsible for certain reliability properties to the services it provides for communication. In short, DARM needs Spread for communication support.

Spread is developed by C and the latest version, Spread 4.1, is used in the work. It contains two parts in Spread, daemon which is used to forward messages between Spread clients and library, libspread, which is used to developing clients. The library contains functions that are able to connect to the daemon and communicate with other Spread clients.

In this paper, each host in the network must have one Spread daemon running on top of it which is a typical phenomenon in a Spread network, however, it is not required that there should be a Spread daemon running on each host in the distributed system.

A Spread network are always contains several host as seen in Figure 2.1, each of them has a Spread daemon running on it whileas the Spread Client, "Client" in the figure, get connection with it through functions provided by libspread. There is usually one Spread daemon on each host but it can be several Spread Clients connect to the same Spread daemon. Within the same network, a Spread daemon communicates with its neighborhoods daemons by utilizing broadcast or multicast

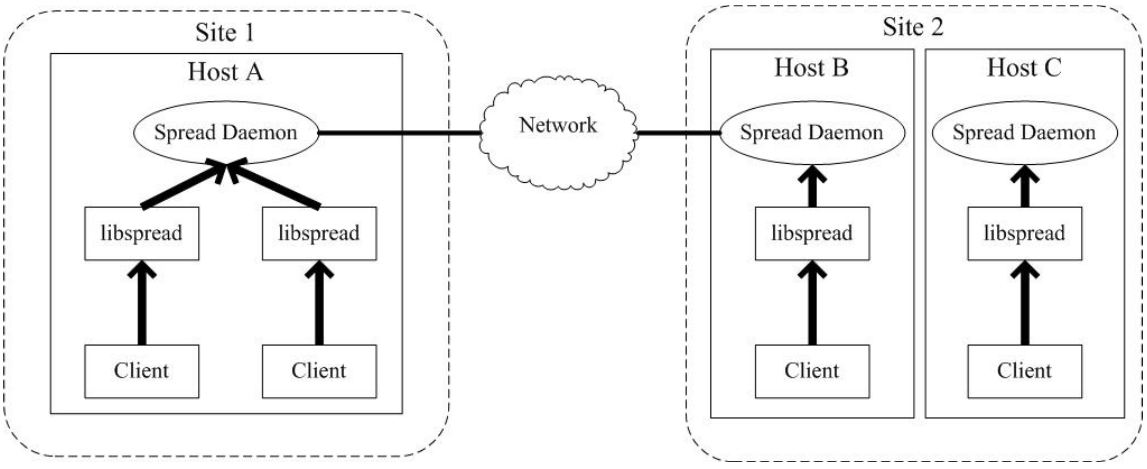


Figure 2.2: Spread Daemon's communication between Sites

traffic. Like the Host A and Host B in Figure 2.1, to use broadcast or multicast in Host A to communicate with Spread daemon in B depends on the capabilities of the LAN environment that A and B situated in.

A real distributed Spread network may have the possibility to span several networks. Each sub networks will have the daemons inside of it communicate with each other as discussed above while the communication between two Spread daemon which locate in two different network segment, like Host A and B in Figure 2.2, will be able to made by unicast over a WAN connection. In this case, a master daemon in each sub network, as Site 1, 2 in Figure 2.2, will use unicast to send messages to other network segments/sites and this master will present on behalf of all daemons locates within the same segments/sites as it is to transmit all messages received from other segments/sites onto its local one. Like as Host B in site 2 shown in Figure 2.2, it will manage messages sending out and receiving on behalf of itself and Host C in Site 2 with another master daemon, Host A, in Site 1. Unicast is also responsible for the communication between libspread and daemon and the library, libspread, is responsible for connection to daemons locates on machines/hosts.

If a master daemon, let's assume Host B here in Figure 2.2, crashes or failed, that each daemon in the same sub network with Host B will be able to take over the responsibility that Host B used to have, which means that each sub network should be access to the internet rather than only master daemon does.

2.2.2 Architecture of DARM

There are a lot of terms and definition defined or used in DARM. The machines or computers that connected with each other in a distributed system through a network are called nodes or hosts and different set of services like application services will be hosted in the form of replicas on top of those nodes or hosts. A collection of such nodes can be comprised of one or more subsets that each subsets, also known as segments, sites or partitions, will contain one or more node. The partitions present the different geographic area of the network and each partition will have all the nodes/hosts inside of it within an identical local area network (LAN). Different type of replicas, which on behalf different of services, may run on the same node however two identical types of replicas should not be able to running on the same node because it will increase the risk of this service if this host suddenly failed which was something that DARM trying to avoid to. Also it is important to notify that all the clients in the following paper refer to the Spread/DARM client which is service replicas or replicas as well.

Figure 2.3 presents a brief deployment in DARM. As it is mentioned above, a Spread network can span into several partitions, in DARM such as site 1 and site 2 in Figure 2.3; and there might be several hosts inside of each site, such as Host 1, 2, 3 in site 1 and Host 4, 5, 6, 7 in site 2. Each host may have different types of replicas running on top of it, such as Host 3 has replica type A and type B running on top of it.

A factory is required on each host in DARM because factory is the one who make installation of service replicas available upon requests, as well as the checking availability of hosts and tracking load on the local machine. All the individual factory that locates on each host in the system get into a factory group and one of the factory will become the factory leading of all the factories like the factory on Host 2 is the factory leader in Figure 2.3.

Those replicas that provide the same service are point towards the same service group, such as A1, A2 and A3 are the replicas for service A while as the B1, B2, B3 and B4 are for service B. The same type of replicas getting into a group which providing an identical service to the client called a service group, like as the Service A group and Service B group in the figure. Each service group should have a minimum service/replica number as the sites, which means the service/replica number a service group holds will be equal or greater than the site number, such as the situation in Figure 2.3: two service group and two sites. It is to be noted

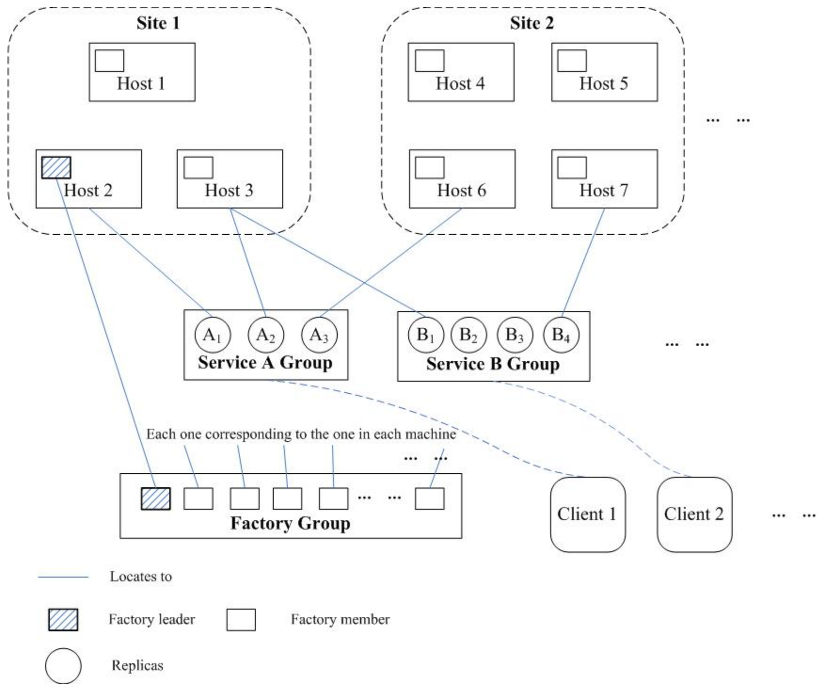


Figure 2.3: Brief deployments in DARM (adapted from [10])

that same type of replicas should not be locate to the same host, e.g. A1 and A2 should locate in two different host, Host 1 and Host 2 in this figure, other than the same one.

As already mentioned in the previous content, DARM is running on top of Spread, while Spread has a library, libspread, DARM also has its library, libdarm. Intended to replace libspread from a client's view as libdarm is, it links to the clients on top of libspread to achieve goal of autonomous operation, as shown in Figure 2.4.

It's easy to see that the a DARM client-spread daemon relationship differs from a Spread client-spread daemon one from adding a libdarm 'layer', that is the DARM library, between the client and the libspread and this library is presenting on half of DARM upon a lot of issues such as removing redundant replicas, initializing required system reconfigurations, collocating replicas etc.

A factory is required to run on each host in DARM and it is an individual separated programme that usually starts after running Spread but before replica services due to its dependency to Spread Daemon. Factories are responsible for the following tasks: creating new replicas; keep tracking the running/alive factories

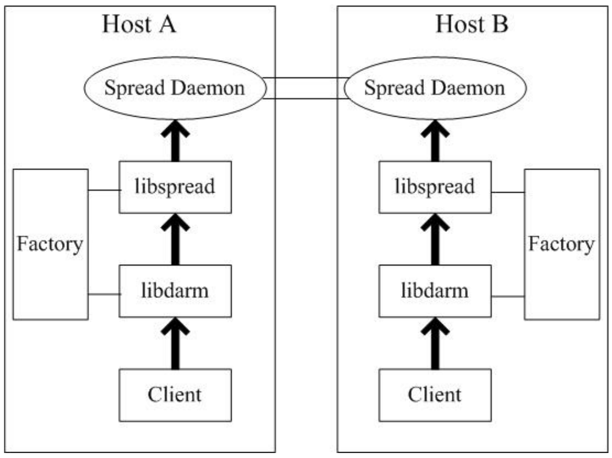


Figure 2.4: DARM Client-Spread Daemon relationship

and their geographical location of the network sites; keeping track of CPU load of machines and the availability of them. The factory leader, also known as factory master, is the one who is responsible for making decision on where (which site) and whom (which host) a new replica should be located on and this decision is made based on the information factories get.

The relationship between libdarm, the library of DARM, and factories in DARM is that factories/factory group will receive requests of creating replica from libdarm and libdarm attaches information such as hosts that running a replica and the living replica numbers in each site to help factory making decision on locating a new replica.

2.2.3 Replica creating in DARM

When a host crashes or something alike happens, a replica needs to be created in order to compensate. libdarm is the one who raise a replica request to the factory group, which is the whole number of factories locates in different hosts, on demand and factory leader will make decision on which (which site) and whom (which host) a new replica should be located on, together with the creating request, some attached information which help factory make decision for the creating mentioned above will be sent to all factory as well, as shown in the left dotted pane in Figure 2.5. All the attached information, that is the currently running-replica host and alive replica numbers in each site, will be considered as active configuration

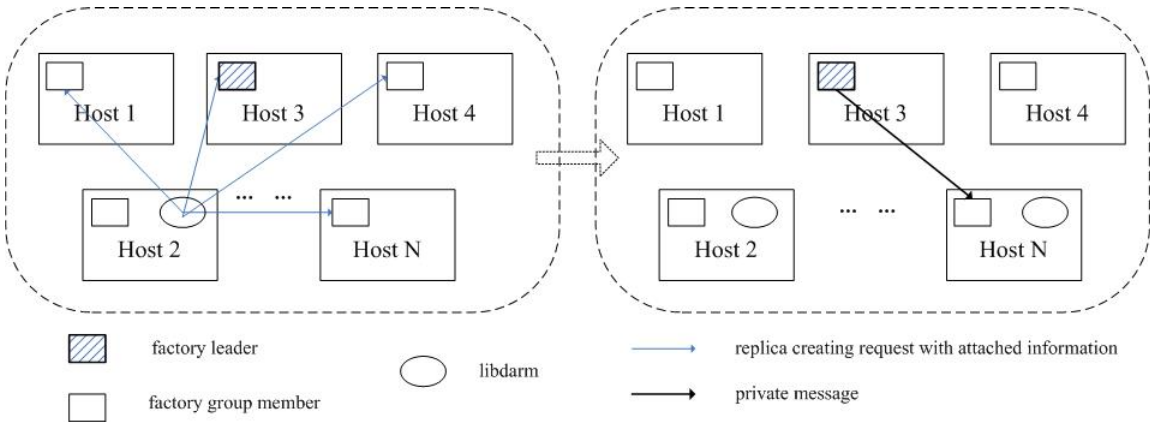


Figure 2.5: Replica Creation

and kept in each factory member in the factory group because every factory has the chance to be selected as a new factory leader if the current one suddenly crashes/fails.

After the factory leader decided the optimal host to locate the new replica which is calculated according to a static priority list, it will send a private message directly to the factory running on that host to realize the request. As the right dotted pane in Figure 2.5 indicates.

A replica placement policy, which is implemented in the factory, is ruling the selection of optimal host to set new replica onto. The main idea of this policy is to find a site with the least number of replicas of a certain type while, at the same time, select one host in this site with the least loading and has no such service running on top of it already. Following this policy, the number of replicas in each site will be balanced distributed so that the possibility of recovering will be lessen when a network partition happens, also two replica of an identical service location on the same host is avoid as it should be, as well as the 'least-burdened' host will be preferential selected when it meets the foregoing criteria.

2.3 Redundancy Scaling

In DARM or other similar framework/system, replicas are always spread over a geographical area and connect with each other within this network as a method to survive from a failure of one replica independent of the others. The number of replicas is varying from one to another due to the different scale of the network and the whole service loading it sustains. Usually, new replicas created at the time when existing replicas are not enough to hold the current loading which is in order to balance the service loading distributed all over the network, however, the service loading is always changing in practice which requires an adjustment to modify the numbers, and that's why a redundancy scaling is to be made to DARM.

As explained above, the redundancy scaling in DARM is about the replica controlling corresponding to different incoming cases. The purpose to do redundancy scaling is to use the resources over the network in a more efficient way, and control replica creation, placement, removing etc. as it is planned.

2.3.1 Policy

1) The fault treatment policy A fault treatment policy united with each service in DARM aimed at scaling the expected service redundancy level. Its objective is to sustain service availability in all sites/partitions, as it is described in Figure 2.3, the replica number in each service group, such as service A group, service B group, should be at least equal to the sites number, just as how it is called, Keep Minimal in Partition. This policy is implemented in the DARM library, libdarm, by utilizing factories to create replacement replicas upon requests while a maximal and minimal redundancy level for services is required as input of this policy.

2) The Replica Remove Policy It would become messy if replicas only be added to a system, thus a proper removing of excessive replicas is quite necessary. The excessive replica exists due to the addition of replica when a partition happens. To be specifically, when a distributed network has become apart/disconnected from each other due to some reason, a fault treatment mechanism will install more replicas in some of the sub networks/partitions in order to keep every sub network/partition a minimal replicas to meet the minimal redundancy level; Once the disconnection has been solved, that is the parted partition merged back to the network, it may occur the situation that the total amount of replicas of an identical service has been far more than the maximal redundancy level which indicates a removing action to be processed. This remove policy is achieved in the DARM library, libdarm.

Figure 2.6 describe the whole procedure alone with replica removing after partitioning, assume a network contains six hosts as Host 1-6 in the figure and three of

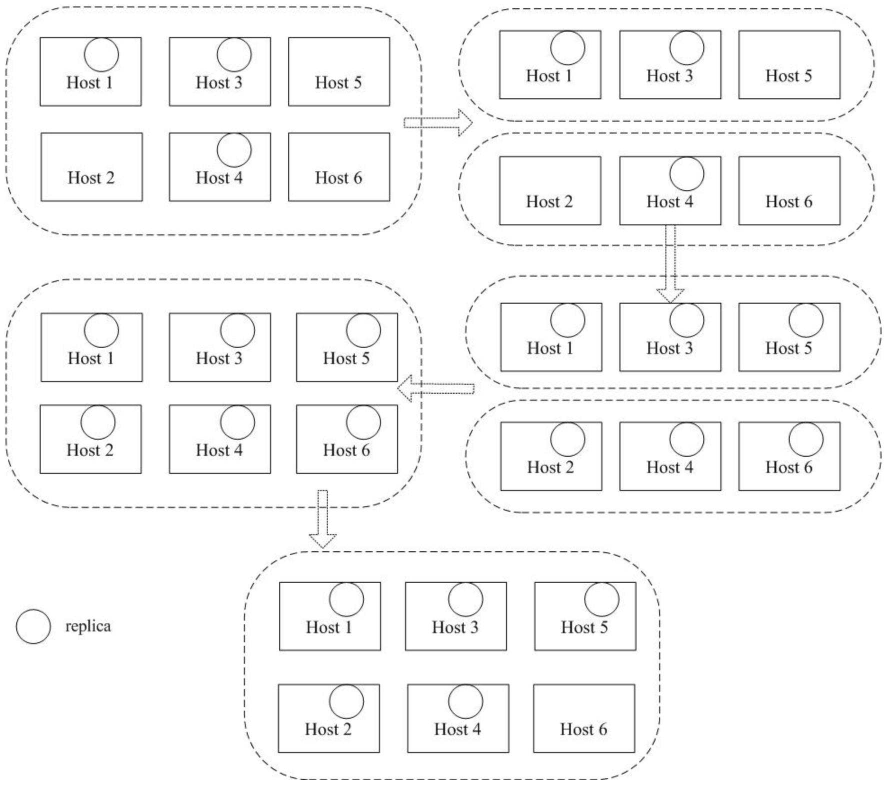


Figure 2.6: Replica creating after partition and removing after re-join

them have a certain type of service replica running on top of it, for this network, the minimal replica redundancy level is 3 while as the maximal is 5. For some reason, a network disconnected and become two smaller networks, which is Host 1, 3, 5 and Host 2, 4, 6, together with the replicas running on top of those Hosts as before disconnection. Then, it is discovered that the minimal replica redundancy level hasn't been reached at both partitions, which is not allowed in the system, so that each of them will create more replicas in order to fulfill the minimal replica redundancy level which is three replicas in each partition; Later on, the disconnection has been solved so that two partition can merged back together as it was and it can be easily seen that the difference between origin state and re-join state is that after re-join, there are more replicas in the network and it is more than the maximal redundancy level, here is 5 in Figure 2.6, which is not allowed neither, at this time, libdarm will detect excessive replica and remove it. libdarm will only remove one replica each time, which means if there are more than one excessive replicas exists in the system, libdarm will remove it one by one.

2.3.2 Static redundancy scaling

As it is mentioned in different policy above, libdarm has the maximal and minimal redundancy level to control the number of replica within an expected range. The maximal and minimal redundancy level are required input in the static redundancy scaling in DARM and the value of these maximal and minimal redundancy level are fixed in the system.

When a client service initials in DARM, libdarm will gradually add replicas to the system to achieve the minimal redundancy level, once this is achieved, the replica number will be back and forth between the minimal and maximal redundancy value. The purpose for setting a lower and higher redundancy level to the system is that if a partition happens to the network, some replicas will be created in order to maintain the client services in each partition, however, those new added replicas will not be removed if a redundancy level is not performed, so that when a partition comes next time, some more replicas will be created again as a cycle begins, thus more and more replicas will be created but never removed in this system which will bring a big chaos to the system or lead it to a messy situation. Similarly, if a partition happens to a network, client services will not be maintained so as to provide to users if it doesn't meet the qualification that there are replicas in each partition, thereby, a minimal redundancy level is required to force a least number of replicas exists in the system.

DARM uses replicas to cope with the risk of error/failure that happens to the system so that the administration to replicas is crucial, redundancy scaling in DARM plays an important rule to help controlling replicas as it is expected to be accordingly.

2.3.3 Dynamic redundancy scaling

What is implemented in this paper is the dynamic redundancy scaling, compared to the static one discussed above, it is aimed at adjusting the living replicas spread over the distributed system accordingly. The reason that the dynamic redundancy scaling presents is the system loading of the client services are varying from time to time, a pre-set or a static redundancy level could hardly fulfill the requests of various changes happens upon a real situation, a better way to always arrange expected and proper replicas over the distributed system is to make the adjustment alone with the system running which is the utilization of dynamic redundancy control.

The principle for a dynamic redundancy scaling is to arrange a right number of replicas according to different situations: when the loading of an identical service on a host become increased, it raises the risks and damages if a sudden failover happens to the host, which creating more replica on some other host will help relieve the 'burden' that carry on the foregoing host; when the loading of a type of service is observed low, it occupies spare resources more than it necessarily need, which removing some low-loaded replica among the existing ones will save and prevent recourse being used indiscriminately.

The element or the factor that effects the dynamic redundancy scaling may be various choices, like as the CPU loading of each host in the distributed network, the network loading or combination of both. Those elements/factors are the reason replica's number changes.

Although a redundancy scaling can be dynamic, it doesn't mean that it will lose control, since the maximal and minimal redundancy level is adjusting accordingly as well in order to range the number of replicas within a reasonable area. Also, to decide details like where a new replica should be created, which replica should be removed, and is one being created/removed will help the whole system etc. are very important in a dynamic redundancy scaling mechanism as it needs to perform properly along with the changes happens to the system.

Compared to a static redundancy scaling, dynamic scaling is more flexible and has more efficient use of the network resources. The drawback of static redundancy scaling is that the lower and higher threshold is static pre-set, it will not easy to estimate a proper value for both of it before actually running such service in the system, even if it has been tested and adjusted already, the situation in a real time performance varies from time to time, it may have a plenty of service requests at the a time while it could be evenly distributed along the system running period, and it is not always the same; In this way, a dynamic redundancy scaling stands out its strengths, it adjust the number of replicas according to the current situation which means no matter how the pre-defined minimal and maximal level is, it could always make modification base on the current demands. This adjustment will not be performed in a sudden way but will be done gradually which means no matter creating or removing replica, it always processes one after another so that the trend of the loading will be balanced follow a smooth curve.

However, what the static and dynamic redundancy scaling in DARM has in common is that they both have a very good schema on removing replicas. Removing replicas is not as adding replicas, once a request is arrived will a new replica be added, it needs more analyze on whether a replica should be removed or not, e.g. if a replica is the only one replica exist, then it shall not be removed.

Chapter 3

Implementation

3.1 Overview

This dynamical redundancy scaling is implementedin standard C, the apparent reason is that the previous work of DARM and its communication toolkit Spread toolkit were implemented in C while the underlying cause is that C is a language which deals with memory directly, it has better performance and effect on modifying, editing other programmes.

The dynamic redundancy is measured on the loading situation of hosts all over the distributed network. In another word, it is aimed at maintaining a proper and befitting number of replicas of a client service by adjusting the maximum and minimum redundancy level base on observing the CPU load of current host-group members.

The main principle behind this redundancy scaling is to add more replicas when high loading of CPU is observed while remove excessive replicas when the CPU load is low. The purpose of introducing this redundancy scaling to DARM is that it will always keep a balanced working environment of loading over all group members in DARM which certainly prevent host from a heavy-load failover and, at the mean time, take full advantage of resource in the system.

Compare with a static redundancy scaling, there are more flexibilities added to the DARM in a dynamic one. Not only will the alteration of replicas upon partition be covered, but the situation under a relevant higher/lower loading of host has been taken care as well.

3.2 Periodical collecting of CPU loading

In DARM framework, it is able for factory to updating and storing the CPU load of the host that the factory running on top with. As mentioned above, this CPU load value is concerned as one of the 'attached' information which will help factory leader make a decision on where and which host will be set a newly created replica. A unix system command, loadavg, is used here in order to read an average load of the last 5, 10 and 15 minutes of the CPU. These three CPU load value are stored in an array named double_load in a structure called darm_client defined in common_groupManagement.h.

In DARM, any change of the current group membership, also described as the change of the view/group view, will lead an updating of the CPU loadings of all the factories in the network, like as, exchange of a new factory leader, a newly created/existing replica join-in/leaving, a failed host, etc. It is also happened when initializing factories on hosts, each factory will update its CPU load value, store it into an array and multicast it as a message to the factory leader. This broadcasting updating load function is factory_send_update_loadavg which is defined in the factory_messageSenders.c and triggered in factory.c. When factory.c being executed at the initialization of starting a factory on host, this function will be triggered to carry out its duty.

When the updating-load message has been received by the factory leader, a _loadAvgMessage function will process the content of received message, pick up the CPU loading value and store it in another array called load.

In order to make this message multicasting and processing after receiving such message, an always true loop is used here to invoke factory_send_update_loadavg periodically. However, it could not just execute this loop in the original thread, since this periodical CPU load reading requires a separate thread to be in charge of it. This thread is defined as 'thread' in factory.c and can be started by start_cpu_update_fork. This start_cpu_update_fork function will give a feedback after starting this thread. Listing 3.1 shows the periodical call of the CPU load value in factory.c.

Listing 3.1: Periodical Reading of CPU loading

```
void thread(mailbox mbox) //define thread
{
while(1)
        {
        factory_send_update_loadavg(mbox);
        sleep(3);
```

```
         }
}
int start_cpu_update_fork() //execute thread
{
         pthread_t id;
         int ret=0;
         ret=pthread_create(&id,NULL,(void *) thread,NULL);
         if(ret!=0)
         {
         DEBUG("Create pthread error!\n");

         exit (1);

         }
         return id;
}

         printf("starting cpu update fork ....\n");
         if(start_cpu_update_fork(mbox)==0)
         printf("cant start cpu update fork\n");
         else
         {
                 printf("cpu update thread start!\n");
         }
```

3.3 Dynamic Controlling

As it is mentioned, the dynamic redundancy is aimed at a more flexible controlling of the replicas numbers in DARM, which has a very tight connection with the usage of native method to the system, that's why a standard C language programme can implement it better than other languages.

In factory, the group members are all factories in the network and they use broadcasting message to contact with each other; similarly, in libdarm, the group members are replicas and there are messages broadcasting in libdarm as well. In this dynamic redundancy scaling, a message aimed at reading the CPU load value from the host where replica locates sent out is required, and it is very similar with how factory read the CPU load from the host.

Besides, a mechanism on how it is actually raise up or draw down the redundancy level is also necessary and the relationship between redundancy levels and the CPU load is also required in the dynamic redundancy scaling implementation.

3.3.1 Message handling in libdarm

libdarm is the library of DARM, it is not only providing the functionality that available from Spread, but configuration handling in DARM as well. Functionalities are distinguished by public and private; public functionalities refers to those functions that available for the application-developing usage while the private ones refers to the functions that for the internal or native method usage. The former one was from the libDarm.c and the later one was from the libDarm_private.c.

In libDarm_private.c, private function such as replica creating, group initializing, message handling, etc. are provided. However, it doesn't handle anything from the memory reading which is important in the dynamic redundancy scaling.

To achieve CPU reading in libdarm, it needs two threads to control two different tasks: one thread is responsible for a periodical reading of the CPU load from the machines that replicas locate on while the other one is in charge of making adjustment for the redundancy level base on the change of CPU load responding from those machines. The second thread will be mentioned in 3.3.2 while the first one is similar to what was implemented in factory. In listing 3.2, the threads were presented, and the start_cpu_update_fork function starts a thread called cpu_thread which will broadcast a message send_update_loadavg periodically.

Listing 3.2: Thread for reading CPU loading in libdarm

```
void cpu_thread() // define cpu_thread
{
while(1)
        {
        send_update_loadavg();
        sleep(3);
         }
}
int start_cpu_update_fork()
{
        pthread_t id;
        int ret=0;
        ret=pthread_create(&id,NULL,(void *) cpu_thread,NULL);
        if(ret!=0)
        {
        DEBUG("Create pthread error!\n");
        exit (1);

        }
        return ret;
}
```

The send_update_loadavg function plays the same role as factory_send_update_-loadavg does in factory which is multicast a message to all group members aimed at reading the expected CPU loading, shown in Listing 3.3. Noted that, this requirement message will not be sent to factory leader but libdarm group members, here refers to the whole replicas. The reason for changing the receivers in the message is that factory and libdarm are not synchronized with each other, they both can read the CPU load however they will not effects on each other if one of them has updated their CPU load value 'list'. The only way to make sure that the CPU load value read by factory and libdarm are the same is that they both do the load reading from the memory.

Listing 3.3: Multicasting messages to libdarm members to get CPU loading value

```
void send_update_loadavg() {
    double loadavg[3];
    char msg[128];
    int msg_length;


        if (getloadavg(loadavg, 3) != 3) {
        ERROR("Failed to retrieve loadavg");
    }

    msg_length = snprintf(msg, 128, "%.2f %.2f %.2f", loadavg[0],
        loadavg[1], loadavg[2]);
    if (msg_length > sizeof(msg)) {
        ERROR("Straaaange loadavg sizes");
}

    SP_multicast(*libdarm_mbox, RELIABLE_MESS, libdarm_group,
        DARM_LOADAVG, msg_length, msg);
}
```

Since each type of replica could only be locate one at most on each host, so that the CPU load feedbacks from all the replicas represent the CPU load from different host in the network, which means there won't be two CPU value coming from the same one.

When the initialization of the DARM group starts, two threads discussed above, will be started along with the initializing, see listing 3.4. In this way, the sending out message on demand of CPU load has been completed.

Listing 3.4: Starting thread in libdarm

```
int res= _Darm_dynamic_control_thread();
        if(!res)
        printf("dynamic control thread starts up!\n");
        res=start_cpu_update_fork();
        if(!res)
        printf("cpu update start up!\n");

    free(message);
```

Once the replicas, that is the libdarm group members, receive such request message, it will response it by invoking function _loadMessage, and it is defined as a 'case' named as DARM_LOADAVG in the function _DARM_Handle_Message in libDarm_private.c. This _DARM_Handle_Message function verifies a series 'cases' and gives specific action command to each case.

Function _loadMessage, shown in listing 3.5, presents the operation upon a CPU loading value required message which will write the CPU load into another array called load, just as how it does in factory that discussed in 3.2.

Listing 3.5: Operation upon requests of CPU loading value

```
void _loadMessage(char *sender,char*mess) {
    darm_client* rm = (darm_client*) group_get_member(sender);
    if (rm == NULL) {
        WARNING("Got LoadAvg from unknown member");
    }

    sscanf(mess, "%lf %lf %lf", &rm->load[0], &rm->load[1], &rm->load
        [2]);
        printf("On message to modify server: %s\n",rm->name);
        printf(" set load[0] is %f\n",rm->load[0]);

}
```

3.3.2 Design of dynamic control

When the message, which contains CPU loading value request, sent out and processed after receiving, it means that the CPU load from different hosts in the network is now available for adjusting the redundancy level.

Tow extern variables are defined in the libdarm_private.h, which is libdarm_min_-replicas and libdarm_max_replicas; libdarm_min_replicas presents the minimal replicas should be exists in a partition while the libdarm_max_replicas presents the maximal ones that one partition could have. That is to say the living replicas exist in a network should be more than or equal to the libdarm_min_replicas and less or equal to the libdarm_max_replicas. The reason to have the libdarm_min_replicas is that each partition will have the opportunity to continue providing client service in case a partitioning happens while the reason for libdarm_max_replicas is that the excessive replicas should be removed after disconnected partition re-joining back together, otherwise the total replica number will be grown rapidly without control.

The architecture of this dynamic controlling effect on the replicas is about the changing of libdarm_min_replicas and libdarm_max_replicas according to an actual CPU loading value, as it is shown in Figure 3.1. It could be seen from figure 3.1 that a pre-set upper threshold and lower threshold is defined in the system in order to keep CPU loading value in a proper and secure range; Once the CPU load value has been retrieved from the host through replicas, it will be put into a comparison with the pre-set threshold mentioned above; For different result of the comparison will lead to a different adjustment on the previous maximal and minimal value. The start sign, '*' refers to the least minimal number of replicas should be maintained in a system, no more replicas should be removed if it reaches this number.

The pre-set upper threshold and lower threshold is chosen as 100 and 0 in this implementation which simulated from a real CPU consuming range. Some rules has been setting in order to make the setting of the upper and lower thresholds: i) the upper threshold should not be above 100 or below 0; ii) the lower threshold should not be more than 100 or less than 0; iii) the lower threshold should not be greater than the upper one, or vice versa. These rules were set according to a common knowledge in logic.

A function called need_modify, presented in the listing 3.6, is utilized to provide a modification value to the original maximal and minimal value. The function

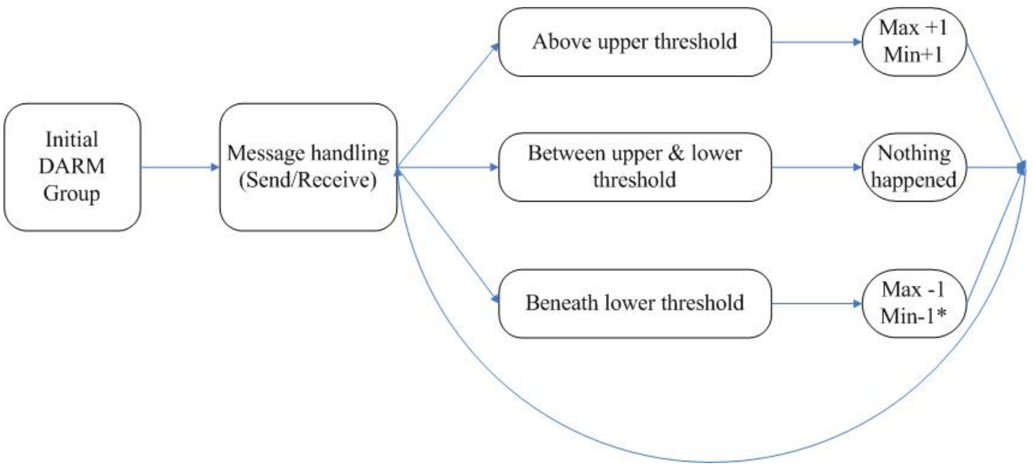


Figure 3.1: Architecture of dynamic control in redundancy scaling

compares the CPU load value which stored in array 'load' with the upper and lower threshold, note that this threshold could be manually defined in the programme. If the load is greater than the upper threshold, a modification value '1' will be returned; if the it is less than the lower threshold, a modification value '-1' will be returned; if it is in the interval between the lower and upper threshold, a modification value '0' will be returned. Note that the CPU load is not a single value but an array which contains three CPU average load of the last 5, 10 and 15 minutes, in this implementation, the latest one, that is the one of the last 5 minutes, will be taken into all forms of calculation.

Listing 3.6: Modification value calculation

```
int need_modify(int load) {
        int retval=0;
        set_upper(2);
        set_lower(1);

        if (load>upper_thresh)
                retval=1;
        else if(load<lower_thresh)
                retval=-1;
        return retval;
}
```

After a modification value calculated from the need_modify function, it will be added to the exsiting libdarm_min_replicas and libdarm_max_replicas in order to modify the replica number to a proper one, and another function, on_modify, is responsible for this task, see listing 3.7. A notice will be printed out on the screen according to different modification value it gains from the need_modify function while at the same time, this modification value will be added to the current max and min replica value, so that a new defined maximal and minimal replica redundancy level will be created since then.

Listing 3.7: Modification on the min/max replica level

```
void on_modify(int modify_val)
{
        if(modify_val==0)
                printf("no need to modify max and minum replica
                   number\n");
        else if(modify_val==1)
                printf("need to add 1 replica\n ");
        else
                printf("need to minus 1 replica\n");
        if ((libdarm_min_replicas<=2)||(libdarm_max_replicas<=4))
                {
                printf("max and min value cant be modified for safety
                     reason\n");
                exit(0);
                }
        libdarm_min_replicas=libdarm_min_replicas+modify_val;
        libdarm_max_replicas=libdarm_max_replicas+modify_val;
        printf("max value is %d, min value is %d\n",
           libdarm_max_replicas, libdarm_min_replicas);
}
```

As it is mentioned in 3.3.1, a separate thread shall be taken responsibility of a dynamic control. This dynamic control thread is implemented as _Darm_dynamic_-control_thread in dynamic_control.c, presented in listing 3.8, as well as another control_thread which send messages to every replica in the libdarm and check their CPU loading by utilizing functions introduced above.

In control_thread, the total number of the group member is required, so that a loop will start from the first one to the last; the loop will exam each group member's CPU loading, once it finds a one bearing a CPU load higher than the upper threshold or lower than a lower threshold, it will jump out of the loop and go to the modification procedure upon the current libdarm_max_replicas value and libdarm_min_replicas value while it will do nothing if every one of them has a CPU load in the interval between the upper and lower threshold. Note that this loop will not wait the entire member, who has a higher or lower CPU loading according to the two thresholds, finds out but will quite to modify the redundancy level as soon as it finds one, this is because that the number of group member, which has a higher or lower CPU load than the threshold, is not concerned as an important matter, but the existence of such group member is the matter that needs to be taken care of.

Listing 3.8: Dynamic control thread in DARM

```
int _Darm_dynamic_control_thread() // execute
    _Darm_dynamic_control_thread
{
        int retval=1;
        pthread_t id;
        retval=pthread_create(&id,NULL, (void*)control_thread,NULL);
        return retval;
}
void control_thread() //define control_thread
{
        while(1)
        {
                sleep(3);
                // send message to every replica
                int number_group=group_get_n_members();
                int load;
                int modify_val=0;
                int i;
                for(i=0;i <number_group;i++)
                {
                        darm_client *rm=group_get_ith_member(i);
                                {
                                        printf("name: %s, load value
                                            is %d\n",rm->name,(int)rm
                                            ->load[0]);
                                 modify_val=need_modify((int)rm->load
                                     [0]);
                                 if(modify_val!=0)
                                         break;
                                }
                }
                on_modify(modify_val);
        }
}
```

The way to name all those functions in this implementation is followed by the rules of how the existing function naming themselves: i) usually an underline '_' was used as a link between words in the same name so as to give more readable information to a function name, such as cpu_thread, start_cpu_update_fork; ii) the prefix '_DARM_' is used to notify a private function in libDarm_private.c, such as _Darm_dynamic_control_thread; iii) if two words are not separatable with an underline in the middle, then capitalize the first letter of the second word, such as _loadMessage. Following the existing rules will not only make newly added code a better integration with the existing one but will also be more convenient for further modification.

3.4 Changes in Replica

The maximal and minimal replica number has been set as a dynamic one in the above implementation which will lead an actual increasing or decreasing of the current living replicas in the network, or it won't change anything of the current situation.

A function called _DARM_Master_Evaluate_Redundancy is in charge of doing creating or killing replicas in libDarm_private.c. A _DARM_Evaluate_Redundancy_-Thread thread is defined so that _DARM_Master_Evaluate_Redundancy could be executed. Different situations were discussed in function_DARM_Master_Evaluate_-Redundancy: i) if the group member, that is the living replicas in the group, is greater than the libdarm_max_replicas, a libdarm_redundancy_kill_client value will be set to '1' and '0' if the group member is less or equals to the libdarm_max_replicas; similarly, ii) if the group member is less than the libdarm_min_replicas, a lib-darm_redundancy_create_client value will be set to '1' and '0' if it is greater than or equal to the libdarm_min_replicas.

This libdarm_redundancy_kill_client and libdarm_redundancy_create_client value will be analyzed according to different cases in the _DARM_Evaluate_Redundancy_-Thread. When it is '1' for the libdarm_redundancy_kill_client, function _DARM_Kill-_Client will be invoked to remove one replica; when it is '1' for libdarm_redundancy-_create_client, function _DARM_Create_Client will be invoked to create a replica. When it is '0' for libdarm_redundancy_kill_client or libdarm_redundancy_create_client , nothing will be done to the replicas. _DARM_Kill_Client and _DARM_Create_Client are both implemented in libDarm_private.c in the previous work of DARM.

Because the value of libdarm_min_replicas and libdarm_max_replicas are varying time from time, the result of libdarm_redundancy_kill_client and libdarm_redundancy_create_client will be changing along with them which will lead to a replica creating or removing or nothing happened in DARM, as described in figure 3.1. Since the reading of CPU loading is periodically invoked, this creating, removing, or keep current value of replica will be triggered time by time which exactly achieve the goal that a dynamic redundancy scaling scheme presented in DARM.

Noted that it is safer to add a replica then remove one, sometimes it is not applicable to do so. If the target replica has reached its 'least' number of replicas that should be maintained in the group, then it is not allowed to removed them in this case; otherwise, some partition will lose the capability to provide such client service, which is something that DARM tries to avoid.

Chapter 4

Test and Evaluation

4.1 Testing environment

To test the availability of this dynamic redundancy scaling functionality, twelve machines were used in total, illustrated in figure 4.1. All of them are running together as a cluster in the same local area network. The Spread and factory will be started on each of them in the beginning of any tests. This is implemented by a shell document which executes start-spread.sh and start-factories.sh instead of manually starting both of them on each machine. Noted that, before starting Spread and factory, the port number should be an unique one and be identical in the following places: Spread configuration file, factory.c and start-factory-with-logging.sh. This starting operation will be done on one of the machines among those twelve. If the Spread and factory are successfully started in all of those twelve machines, it will present a notice as it is illustrated in figure 4.2.

All the standard C code implemented to achieve the dynamic redundancy scaling in DARM will be compiled under linux by cmake in order to gain a libDarm.so file and it will be used as a library. A test application used here is the test_service.c which is a simple test method to check the availability of this redundancy scaling.

4.2 Experimental Result

4.2.1 First testing

By running the test_service.c, the result is illustrated as figure 4.3. Two notices given at the very beginning of the output shows that the dynamic control thread

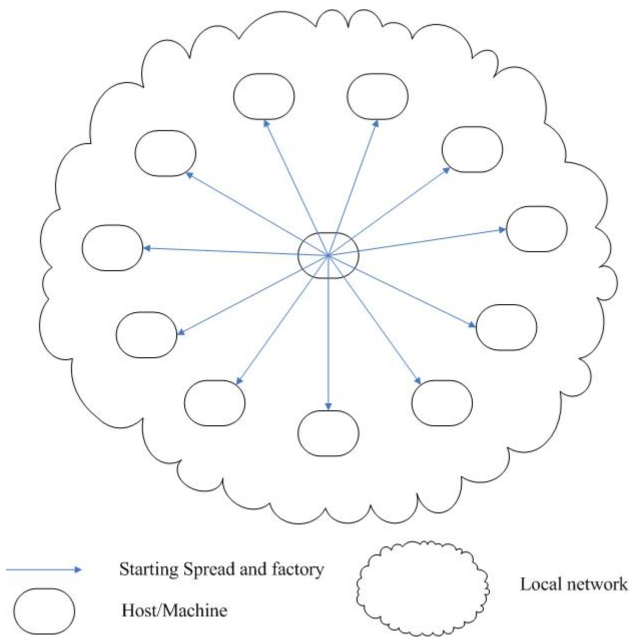
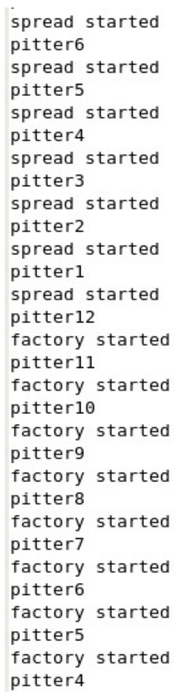
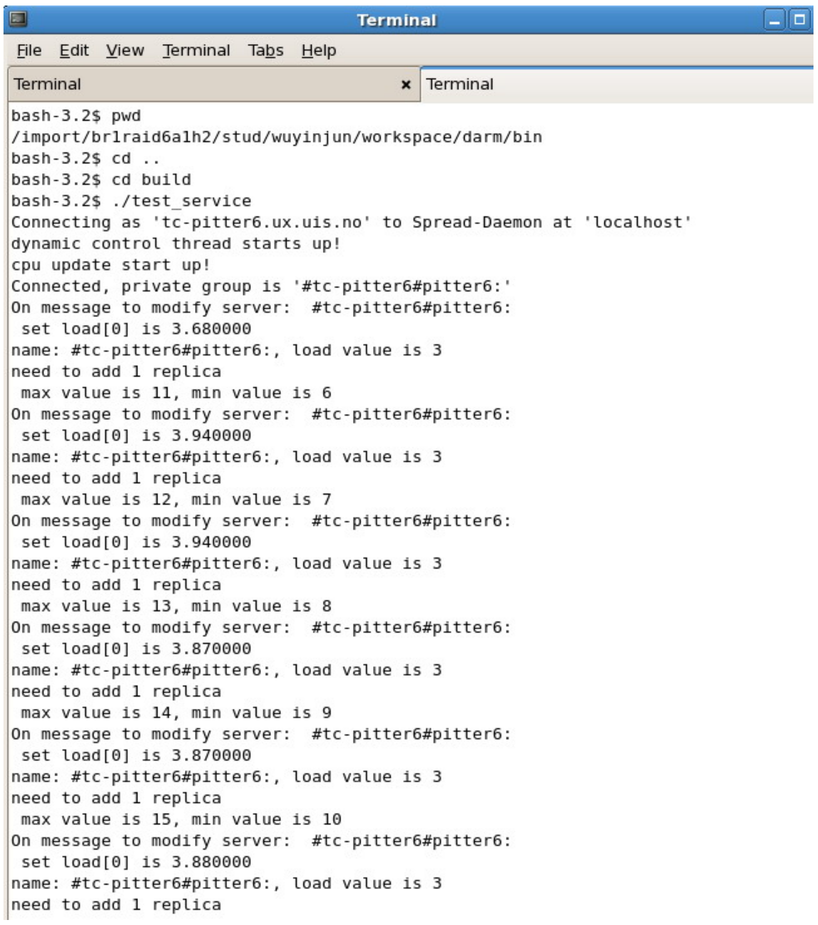


Figure 4.1: Setting up of the test



```
spread started
pitter6
spread started
pitter5
spread started
pitter4
spread started
pitter3
spread started
pitter2
spread started
pitter1
spread started
pitter12
factory started
pitter11
factory started
pitter10
factory started
pitter9
factory started
pitter8
factory started
pitter7
factory started
pitter6
factory started
pitter5
factory started
pitter4
```

Figure 4.2: Starting Spread and factory successfully



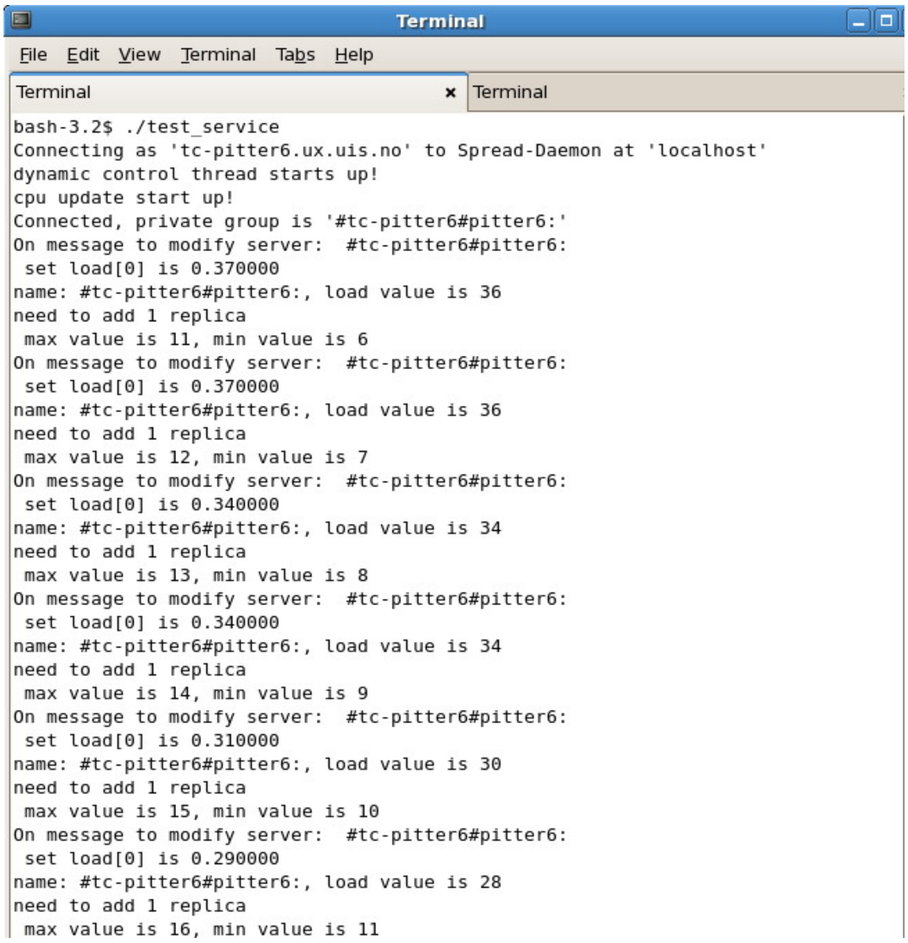
```
bash-3.2$ pwd
/import/br1raid6a1h2/stud/wuyinjun/workspace/darm/bin
bash-3.2$ cd ..
bash-3.2$ cd build
bash-3.2$ ./test_service
Connecting as 'tc-pitter6.ux.uis.no' to Spread-Daemon at 'localhost'
dynamic control thread starts up!
cpu update start up!
Connected, private group is '#tc-pitter6#pitter6:'
On message to modify server:  #tc-pitter6#pitter6:
 set load[0] is 3.680000
name: #tc-pitter6#pitter6:, load value is 3
need to add 1 replica
 max value is 11, min value is 6
On message to modify server:  #tc-pitter6#pitter6:
 set load[0] is 3.940000
name: #tc-pitter6#pitter6:, load value is 3
need to add 1 replica
 max value is 12, min value is 7
On message to modify server:  #tc-pitter6#pitter6:
 set load[0] is 3.940000
name: #tc-pitter6#pitter6:, load value is 3
need to add 1 replica
 max value is 13, min value is 8
On message to modify server:  #tc-pitter6#pitter6:
 set load[0] is 3.870000
name: #tc-pitter6#pitter6:, load value is 3
need to add 1 replica
 max value is 14, min value is 9
On message to modify server:  #tc-pitter6#pitter6:
 set load[0] is 3.870000
name: #tc-pitter6#pitter6:, load value is 3
need to add 1 replica
 max value is 15, min value is 10
On message to modify server:  #tc-pitter6#pitter6:
 set load[0] is 3.880000
name: #tc-pitter6#pitter6:, load value is 3
need to add 1 replica
```

Figure 4.3: First testing

and cpu updating has been started. 'set load[0]' represent the average CPU loading value from the last 5 minutes, as in this experiment, the value is 3.68. This 3.68 is the actual data of the CPU load, not counted as percentage form.

Some initialization values are set as follow: maximal replica number is 11 while the minimal is 6; the upper threshold of the CPU load is 2 while the lower threshold is 1. It could be easily observed that, a new replica is asked to be created since the CPU load is always above then the upper threshold; as a consequence, both maximal and minimal replica number are increasing as well.

Alone with the maximal and minimal replica number increasing, a subtle decrease of the CPU load has been achieve although the decreasing is rather small. In this case, a second testing was draw in order to magnify the consequence in the first testing.



```
bash-3.2$ ./test_service
Connecting as 'tc-pitter6.ux.uis.no' to Spread-Daemon at 'localhost'
dynamic control thread starts up!
cpu update start up!
Connected, private group is '#tc-pitter6#pitter6:'
On message to modify server:  #tc-pitter6#pitter6:
 set load[0] is 0.370000
name: #tc-pitter6#pitter6:, load value is 36
need to add 1 replica
 max value is 11, min value is 6
On message to modify server:  #tc-pitter6#pitter6:
 set load[0] is 0.370000
name: #tc-pitter6#pitter6:, load value is 36
need to add 1 replica
 max value is 12, min value is 7
On message to modify server:  #tc-pitter6#pitter6:
 set load[0] is 0.340000
name: #tc-pitter6#pitter6:, load value is 34
need to add 1 replica
 max value is 13, min value is 8
On message to modify server:  #tc-pitter6#pitter6:
 set load[0] is 0.340000
name: #tc-pitter6#pitter6:, load value is 34
need to add 1 replica
 max value is 14, min value is 9
On message to modify server:  #tc-pitter6#pitter6:
 set load[0] is 0.310000
name: #tc-pitter6#pitter6:, load value is 30
need to add 1 replica
 max value is 15, min value is 10
On message to modify server:  #tc-pitter6#pitter6:
 set load[0] is 0.290000
name: #tc-pitter6#pitter6:, load value is 28
need to add 1 replica
 max value is 16, min value is 11
```

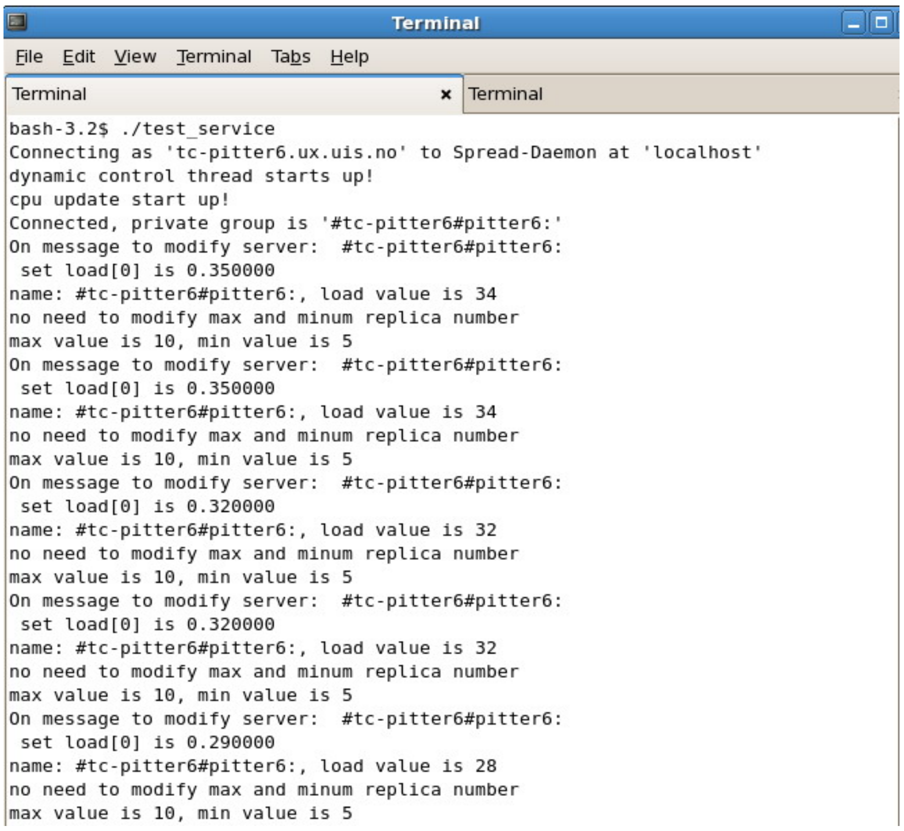
Figure 4.4: Second test with upper threshold 30

4.2.2 Second testing

In order to gain a clearer result, a simple way could be act as to magnify the CPU loading, as well as the upper and lower threshold. To do that, it is just multiply the origin CPU load with 100 and then re-set the two thresholds in a proper way, here the upper and lower threshold are 30 and 20 respectively. The result of running the test_service.c after magnification is presented in Figure 4.4.

Figure 4.4 shows that an original CPU load value is magnifies from 0.37 into 37 which is certainly above the upper threshold 30, so that a new replica needs to be created. Once new replicas gradually created one after another, the CPU loading is slowly dropping down as a result.

In another case, if the CPU load falls in the interval between upper and lower



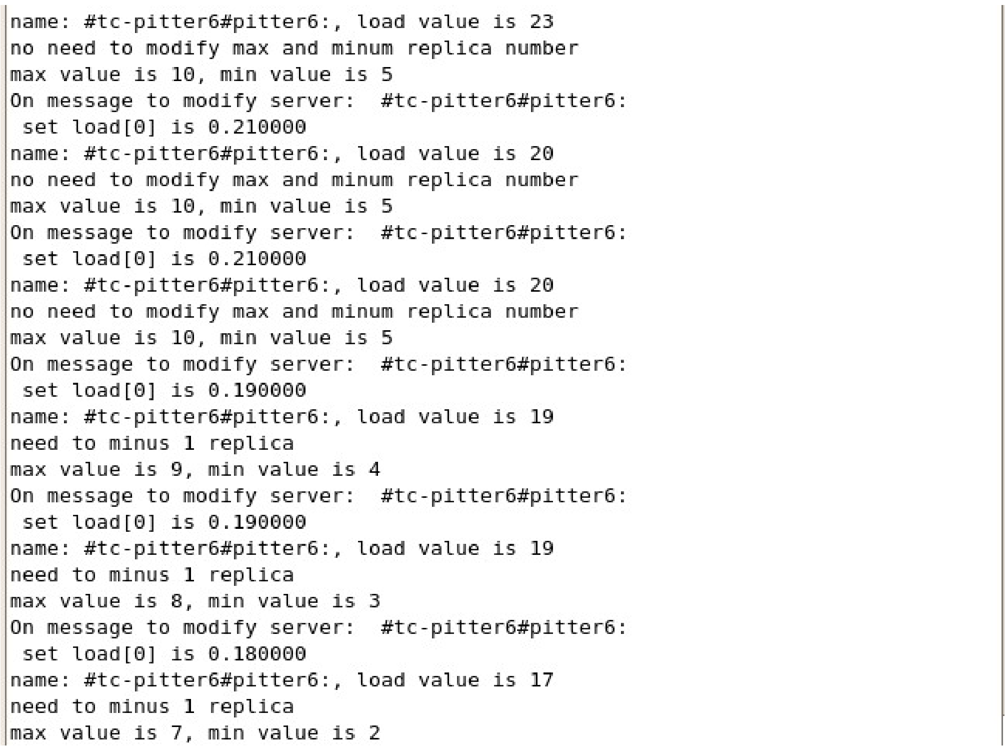
```
bash-3.2$ ./test_service
Connecting as 'tc-pitter6.ux.uis.no' to Spread-Daemon at 'localhost'
dynamic control thread starts up!
cpu update start up!
Connected, private group is '#tc-pitter6#pitter6:'
On message to modify server:  #tc-pitter6#pitter6:
 set load[0] is 0.350000
name: #tc-pitter6#pitter6:, load value is 34
no need to modify max and minum replica number
max value is 10, min value is 5
On message to modify server:  #tc-pitter6#pitter6:
 set load[0] is 0.350000
name: #tc-pitter6#pitter6:, load value is 34
no need to modify max and minum replica number
max value is 10, min value is 5
On message to modify server:  #tc-pitter6#pitter6:
 set load[0] is 0.320000
name: #tc-pitter6#pitter6:, load value is 32
no need to modify max and minum replica number
max value is 10, min value is 5
On message to modify server:  #tc-pitter6#pitter6:
 set load[0] is 0.320000
name: #tc-pitter6#pitter6:, load value is 32
no need to modify max and minum replica number
max value is 10, min value is 5
On message to modify server:  #tc-pitter6#pitter6:
 set load[0] is 0.290000
name: #tc-pitter6#pitter6:, load value is 28
no need to modify max and minum replica number
max value is 10, min value is 5
```

Figure 4.5: CPU loading lay in the middle of upper and lower threshold

threshold, then it will be outline as a 'peace' period since no replica needs to be add or remove from the system, shown in figure 4.5. Besides, due to a large amount of replicas existing in the system, the CPU loading value will continuously drop down smoothly.

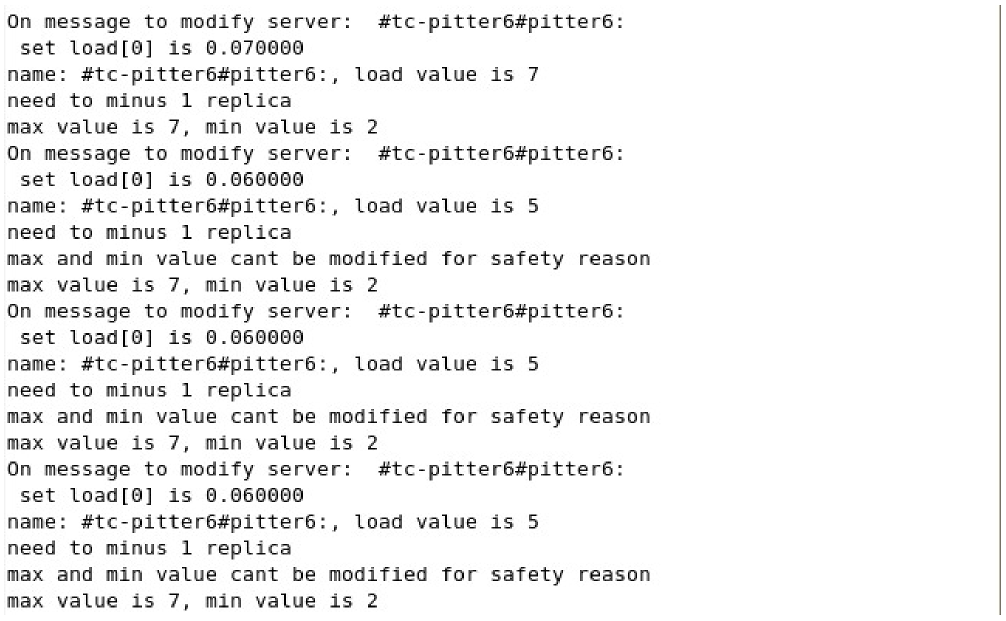
If the CPU load keeps going down, it will be equal or less than the lower threshold 20 in the test, so that excessive replicas will be removed of one each time, see figure 4.6.

Since the minimal replica number is decreasing due to the excessive replicas removing, it will reach the 'least' number of replicas that must be kept in the system, in this test, 2 replicas should be kept at least. Figure 4.7 shows the result when minimal replicas reached to 2, it could be seen that the removing is stopped once the minimal replica number reaches two, at the mean time, the CPU loading tends to stay in a certain range.



```
name: #tc-pitter6#pitter6:, load value is 23
no need to modify max and minum replica number
max value is 10, min value is 5
On message to modify server:  #tc-pitter6#pitter6:
 set load[0] is 0.210000
name: #tc-pitter6#pitter6:, load value is 20
no need to modify max and minum replica number
max value is 10, min value is 5
On message to modify server:  #tc-pitter6#pitter6:
 set load[0] is 0.210000
name: #tc-pitter6#pitter6:, load value is 20
no need to modify max and minum replica number
max value is 10, min value is 5
On message to modify server:  #tc-pitter6#pitter6:
 set load[0] is 0.190000
name: #tc-pitter6#pitter6:, load value is 19
need to minus 1 replica
max value is 9, min value is 4
On message to modify server:  #tc-pitter6#pitter6:
 set load[0] is 0.190000
name: #tc-pitter6#pitter6:, load value is 19
need to minus 1 replica
max value is 8, min value is 3
On message to modify server:  #tc-pitter6#pitter6:
 set load[0] is 0.180000
name: #tc-pitter6#pitter6:, load value is 17
need to minus 1 replica
max value is 7, min value is 2
```

Figure 4.6: CPU loading becoming less than the lower threshold



```
On message to modify server:  #tc-pitter6#pitter6:
 set load[0] is 0.070000
name: #tc-pitter6#pitter6:, load value is 7
need to minus 1 replica
max value is 7, min value is 2
On message to modify server:  #tc-pitter6#pitter6:
 set load[0] is 0.060000
name: #tc-pitter6#pitter6:, load value is 5
need to minus 1 replica
max and min value cant be modified for safety reason
max value is 7, min value is 2
On message to modify server:  #tc-pitter6#pitter6:
 set load[0] is 0.060000
name: #tc-pitter6#pitter6:, load value is 5
need to minus 1 replica
max and min value cant be modified for safety reason
max value is 7, min value is 2
On message to modify server:  #tc-pitter6#pitter6:
 set load[0] is 0.060000
name: #tc-pitter6#pitter6:, load value is 5
need to minus 1 replica
max and min value cant be modified for safety reason
max value is 7, min value is 2
```

Figure 4.7: Minimal replicas reaches bottom

4.3 Evaluation

To evaluate this redundancy scaling mechanism, a 10-time testing is deployed. A twelve machine cluster is used here as it used in the previous testing. After Spread and factory running on each of those machines, the CPU load value of each machine, as well as corresponding the minimal and maximal number of replica, will be recorded and this will be done 10 times with a random CPU consuming service taking alone with each time; this is in order to have an average view of the functionality of this dynamic redundancy scaling. Figure 4.8-4.9 shows the average CPU value over 10-time testing on the 12 machines respectively. The horizontal coordinate represents the time while the vertical represents the CPU value. The time coordinate is presented by single units while each unit is 3 second in this case.

If the maximal and minimal replica number is taking into consideration along with the CPU loading value, it would be presented as figure 4.10 with horizontal coordinate representing the time elapsing and the vertical one representing the CPU value, as well as the maximal and minimal replica number. It could tell from the figure that: 1) When the CPU value is comparatively high, the maximal and minimal number has an increasing trend while a decreasing trend on a lower CPU value. 2) Although the CPU consuming of each service may vary from each other, the CPU value itself has a trend to fall in a lower range after a dynamic adjusting by the maximal and minimal value of replicas for a certain period. 3) CPU value has more obvious changes during the same period that maximal and minimal replica numbers changes little, which means that the replica redundancy level could lower and smoother the CPU value without lose control of itself. 4) The protuberance in the CPU value, which is extra service suddenly adding to the existing one that make CPU value increased, doesn't influence the adjusting effect on the replica numbers since this dynamic redundancy scaling focus more on a general trend which prevent system failover due to a high CPU loading by adjusting replica numbers.

On the whole, the performance of this dynamical redundancy scaling shows that it has effect on controlling the replica numbers in order to prevent a failover of the system because of high loading, at the same time, the replica number itself won't lose control. By introducing this redundancy scaling to the existing DARM framework will help system deploy service replica in a better and proper way while at the same time utilize the network resources in a more efficient and flexible way.

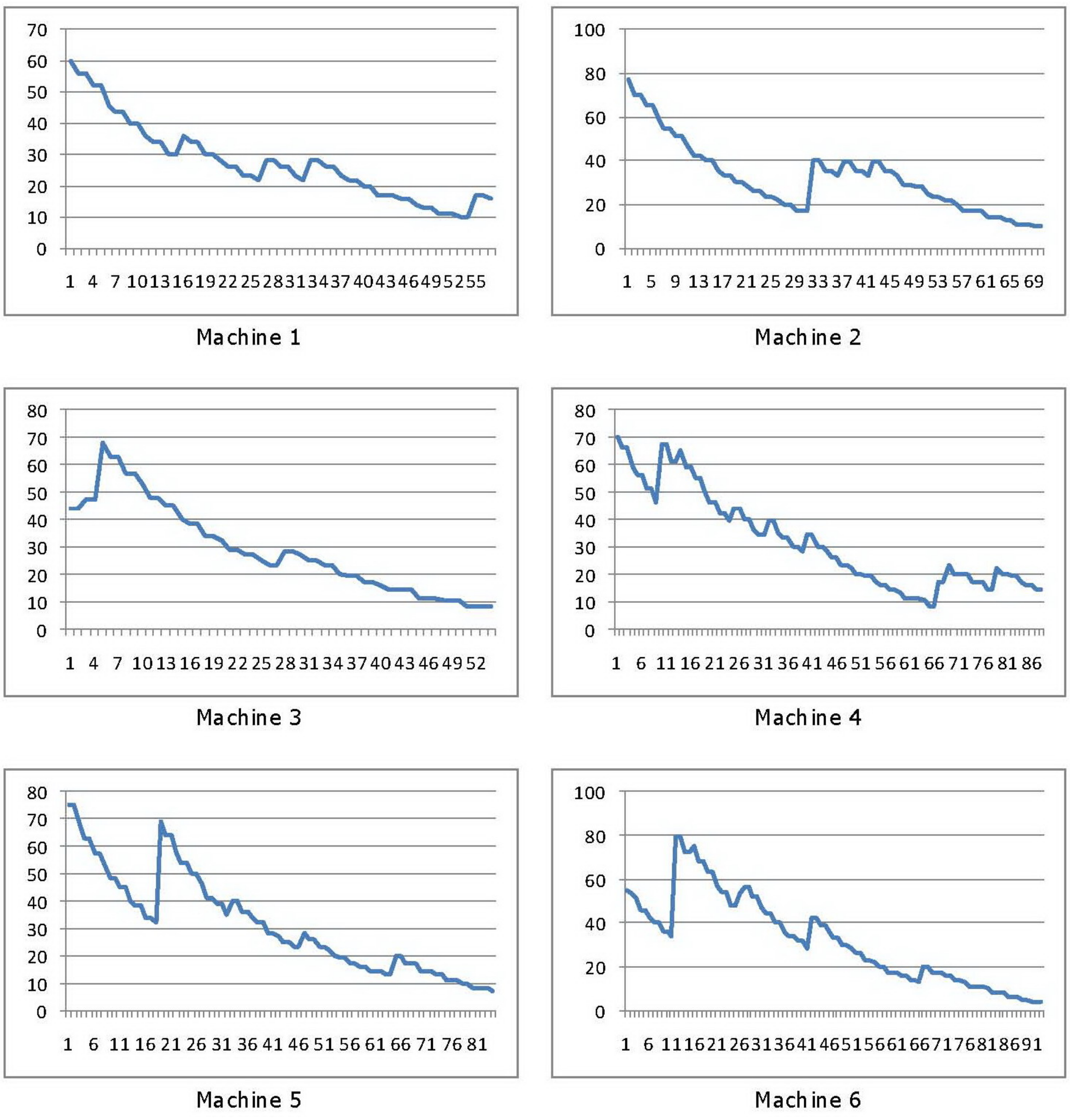


Figure 4.8: Average CPU value on machine 1-6

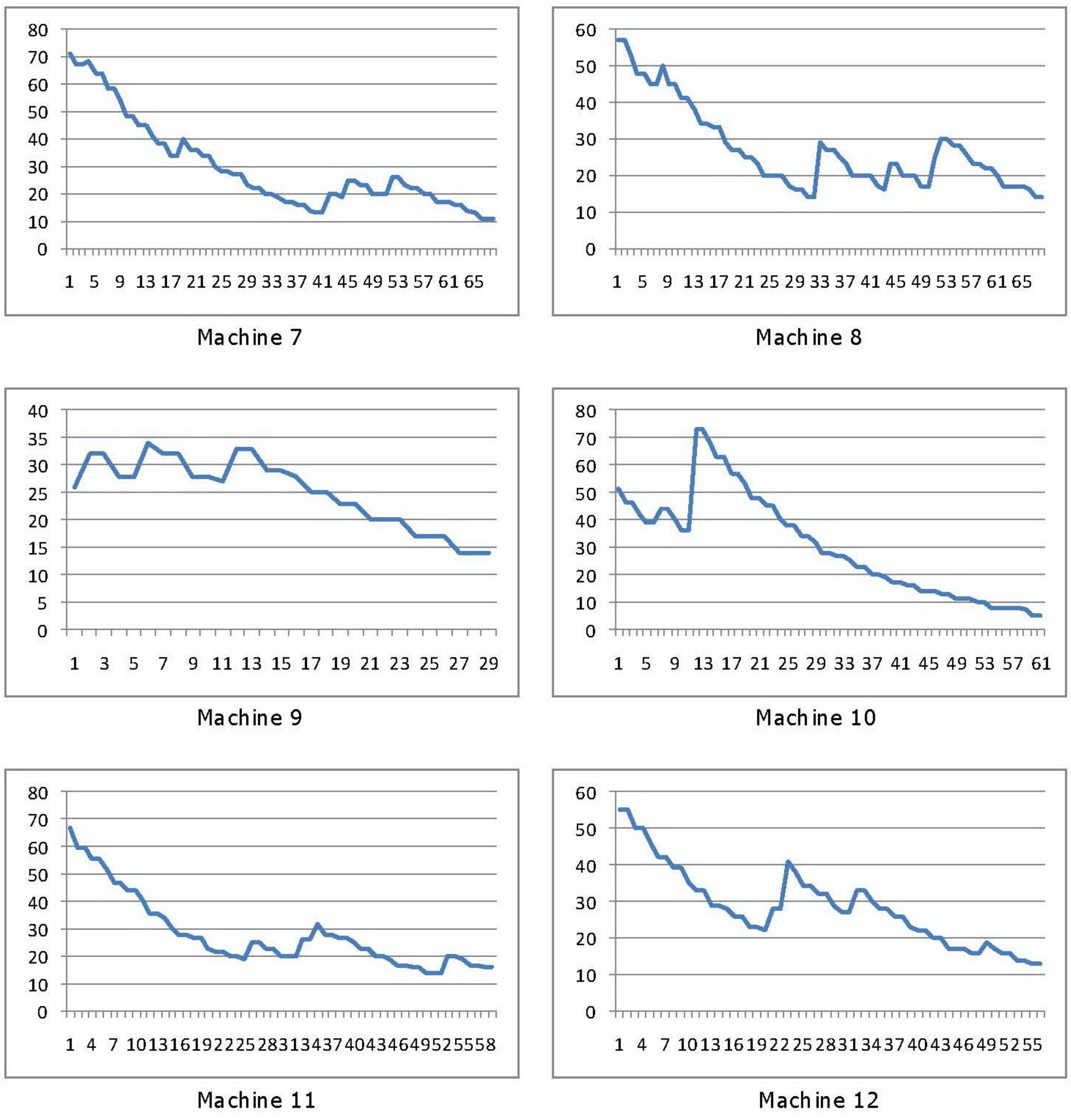


Figure 4.9: Average CPU value on machine 7-12

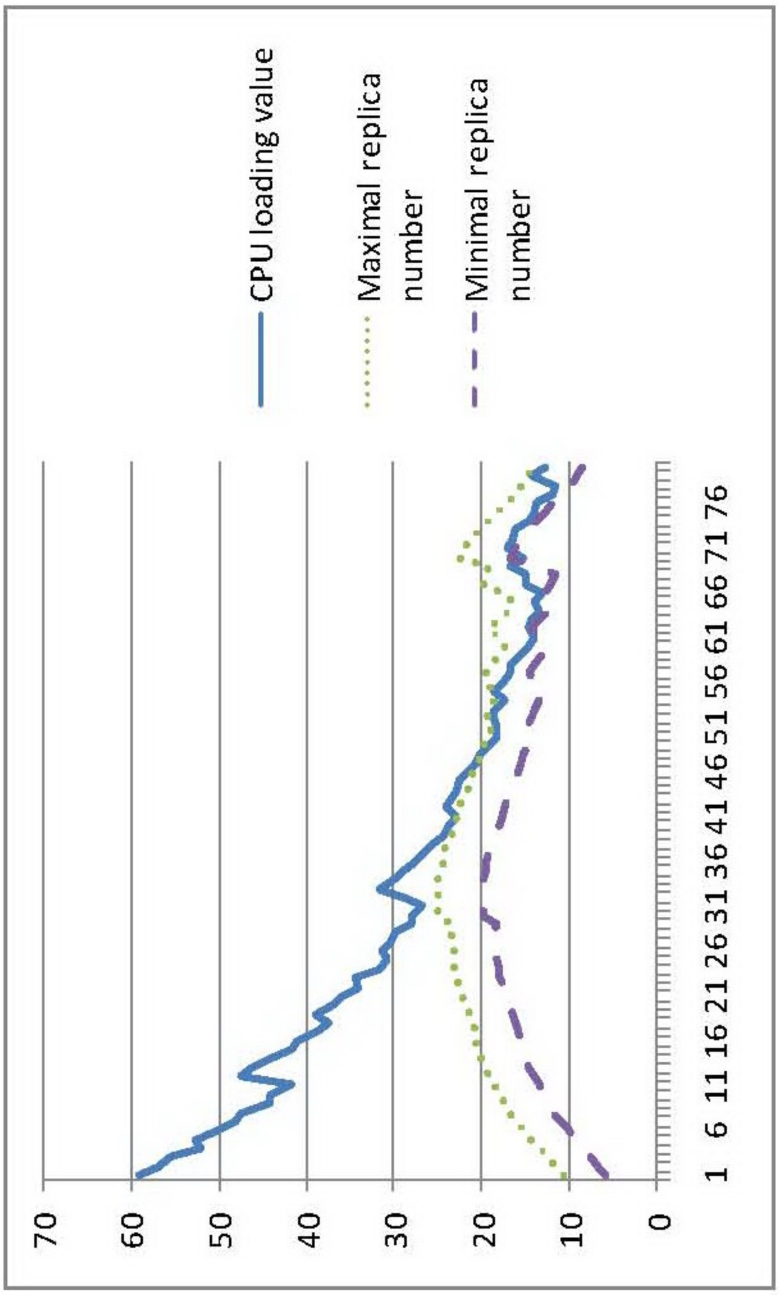


Figure 4.10: Relationship between CPU value and max/min value

Chapter 5

Conclusion

This paper presents the scheme and conceives of a dynamic redundancy scaling in the Distributed Autonomous Replication Management (DARM) framework, which primary goal is to bring out a more flexible and optimal redundancy control mechanism in DARM. This dynamic redundancy scaling is implemented base on the previous work of a static redundancy scaling in DARM which only contains a pre-set and static redundancy level for the system. The consequence of this redundancy scaling is that it will always keep a balanced loading to all members in partition which certainly prevent host from failing because of a heavy loading and, at the mean time, take full advantage of resources in the system. A simple way to reveal the mechanism of this redundancy scaling is to add more replicas when a high CPU loading is observed while remove some replicas when the CPU loading is low.

The testing result of this dynamic redundancy scaling indicates that DARM is able to control the replica spread over the whole network on analyze of the host CPU loading. DARM will perform an adjusting on the current redundancy level when the host suffering from higher loading or taking too little loading, which will of course lead the number of relevant replicas existing in the system increasing or decreasing from time to time. In a way, this dynamic redundancy scaling helps DARM have a more efficiency utilization of the network resource, as well as a better performance.

Chapter 6

Future Work

6.0.1 A network load based dynamic redundancy scaling

The current dynamic redundancy scaling is implement base on the CPU loading of each host; it would be very useful to considering a whole network's loading instead of individual host, or a combination of a network loading and individual CPU loading. A network loading may reflect the request of adding or removing replicas on a group view rather, because in some cases, a loading of the partition can weight more than individual hosts.

To implement it, a policy on how to evaluate a network's loading should be designed and implemented in libDarm_private.c; The network loading value itself could be defined as a part of a particular structure which stores the loading value through a set of system calls on all network members or it could be the result from an algorithm which calculate the network loading base on the individual CPU loading in a group. A rule for handling network value, such as setting an upper threshold or a lower threshold for a comparison to the network loading is also required in the implementation. After all, replicas' creating and removing operation should be connected with the result of certain comparisons.

6.0.2 Different type of service handling

This dynamic redundancy scaling is single identical service oriented, which means all the replicas are of the same type. In the real situation, it always possible that one host has different type of services running on top of it. Because of this, a mechanism aimed providing a dynamic redundancy control while different type of services running in the partition is highly interested.

To implement this, replica needs a name tag or label as part of its structure in order to distinguish from each other; also, to track the CPU loading of each type of replica requires an individual thread so that it could control the starting tracking, making periodical call to retrieve the loading value, etc. Because simpler and smaller client services replicas are usually occupy less CPU loading than complicate and bigger ones, so that when a host has a very high CPU loading in total it need to find out which replica is the one occupies the most of the loading, and then adding new replicas of this type of service in other machines which will make actual contributes on dropping down the CPU loading of this host.

Bibliography

[1] *Distributed networking definition.* website. http://en.wikipedia.org/wiki/Distributed_Networking.

[2] *Fault-tolerant system definition.* website. http://www.bing.com/reference/semhtml/Fault-tolerant_system?mkt)=zh-CN.

[3] *Federal Standard 1037C.* website. http://www.its.bldrdoc.gov/fs-1037/dir-012_1752.htm.

[4] *Spread.* website. http://spread.org/.

[5] Y.A. Claudiu, Y. Amir, C. Danilov, and J. Stanton. *A Low Latency, Loss Tolerant Architecture and Protocol for Wide Area Group Communication.* 2000.

[6] J.L. Gilje. *Autonomous Fault Treatment in the Spread Group Communication System.* 2007.

[7] H. Meling. Adaptive middleware support and autonomous fault treatment: Architectural design, prototyping and experimental evaluation. PhD thesis, Norwegian University of Science and Technology, 2006.

[8] H. Meling and B.E. Helvik. *ARM: Autonomous replication management in Jgroup.* In *Proc. of the 4th European Research Seminar on Advances in Distributed Systems.* Citeseer, 2001.

[9] H. Meling, A. Montresor, B.E. Helvik, and O. Babaoglu. *Jgroup/ARM: a distributed object group platform with autonomous replication management.* *Software-Practice and Experience*, 38(9):885–924, 2008.

[10] Hein Meling and Joakim L. Gilje. *A Distributed Approach to Autonomous Fault Treatment in Spread*. In *2008 Seventh European Dependable Computing Conference*, 2008.

[11] A. Montresor. System Support for Programming Object-Oriented Dependable Applications in Partitionable Systems. PhD thesis, University of Bologna, 2000.