Distributed Name Service in Harness

Tomasz Tyrakowski†, Vaidy Sunderam† and Mauro Migliardi†

†Department of Math & Computer Science
Emory University
Atlanta, Georgia
30302
{ttomek | vss | om}@mathcs.emory.edu

Abstract. The Harness metacomputing framework is a reliable and flexible environment for distributed computing. A shortcoming of the system is that services are dependent on a name service (a single point of failure) where all Harness Distributed Virtual Machines are registered. Thus, there is a need to design and implement a more reliable name service. This paper describes the Harness Distributed Name Service (HDNS) which aims to address this shortcoming. Section 2 outlines the role of the name service in Harness. Section 3 extends this discussion by describing the design of the HDNS. Finally, in sections 4 and 5, we present the services fault-tolerance mechanisms and give our conclusions.

1 Introduction

Harness is an experimental metacomputing framework that is based upon the principle of dynamically reconfigurable, networked virtual machines. Harness supports reconfiguration not only in terms of the computers and networks that comprise the virtual machine, but also in the capabilities of the virtual machine itself. These characteristics may be modified under user control via a *plugin* mechanism, which is the central feature of the system. The plugin model provides a virtual machine environment that can dynamically adapt to meet an applications needs, rather than forcing the application to conform to a fixed model [2].

The fundamental abstraction in the Harness metacomputing framework is the *Distributed Virtual Machine* (DVM). Heterogeneous computational resources may enroll in a DVM at any time, however, at this level the DVM is not ready to accept requests from users. To begin to interact with users and applications, the heterogeneous computational resources enrolled in a DVM need to load plugins. A plugin is a software component that implements a specific service. Users may reconfigure the DVM at any time, both in terms of the computational resources enrolled and in terms of the services available by loading and unloading plugins [1].

2 Name Service In Harness

Each Distributed Virtual Machine (DVM) in Harness consists of exactly one DVM server and an arbitrary number of Harness kernels. If a DVM server crashes, it is automatically restored by one of the kernels. The DVM server is responsible for event propagation and DVM state updates - it is the center of the star (see also [1]) while the kernels can be considered as the branch nodes. New kernels may join a DVM at any

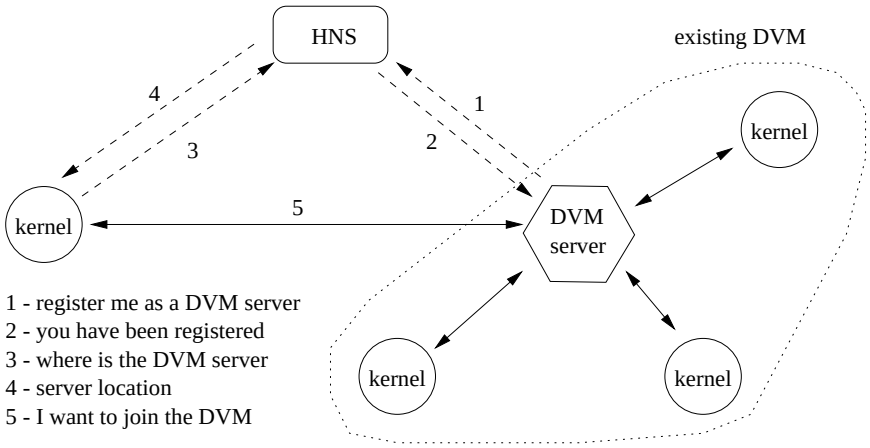

Fig. 1. A New Kernel Joining An Existing DVM.

time, performing the join protocol with the DVM server. The fact that a DVM server is automatically restored is a boon, but it also introduces some complications. New kernels added to the DVM have no mechanism for identifying where the current DVM server is, unless they consult the *Harness Name Service* (HNS) for this information. This service stores pairs of form:

(DVM name, (host, port))

where DVM name is a string unique within the naming service and (host, port) is the location of the server for the specified DVM.

The location of the name server is defined in the *harness.defaults* configuration file, which is present on all machines enrolled in the DVM. Therefore when a new kernel starts, it reads the name server location from the configuration file and then asks the name server for the location of the DVM server. If there is no server registered for the DVM, the kernel spawns a new one.

This strategy implies a specific behavior in the name service. When a DVM server attempts to register itself as a server for a DVM, the name server has to return one of the following:

- registration was successful - assume the role of DVM
- there is another DVM server registered already - the server is (host,port).

In no situation, can a name server refuse the registration of a DVM server, without giving information about the existing DVM server. The name service is also responsible for keeping the list of dormant daemons running, so that any participant can ask about the list of hosts, which can be enrolled into a DVM remotely. A dormant daemon is basically a small Java application, which listens on a specified port and spawns a Harness kernel for a particular DVM when a request comes from the network. The HNS maintains a list of the hosts dormant daemons run on. A dormant daemon also registers itself in the name server after it starts, but this scenario is much simpler - the name server just registers it and inserts the details into its store.

Both DVM servers and dormant daemons are required to refresh their entries periodically, otherwise the name server removes the corresponding entry from its tables. In summary, there are the following types of requests:

- register a DVM server (or return the information about the current server)
- register a dormant daemon
- refresh a DVM server entry (which is equal to re-registering it)
- refresh a dormant daemon entry (equivalent to re-registering)
- give information about the DVM server for a specified DVM
- give a list of hosts running dormant daemons

As was noted above, the name service is essential to the consistent operation of DVM servers and kernels. Thus, a single name server is a single point of failure for the whole DVM. When the name server crashes or is unavailable, new kernels are not able to enroll into a DVM and in the case of a DVM server crash, there is no way to restore it[1]. The name service reliability is critical in this context, and it is impossible to achieve reliability with a centralized name server operating on one physical machine. For this reason, there is a need to design and implement a distributed naming service.

3 The Harness Distributed Name Service

The *Harness Distributed Name Service* (HDNS) is intended to be more reliable provider of the services described in section 2. Initially, it has to consist of more than one physical machine, which immediately implies a number of issues. The first problem is the choice of topology. Another, is synchronization and propagation of data. The ring topology is most suited to our requirements as it is simple to implement. Instead of a single name server, we have a ring consisting of up to 8 physical machines. The number of machines is large enough to provide satisfactory reliability and fault tolerance. This should also keep the ring fast enough to handle client requests and not cause delays. As was noted above, the DVM servers and dormant daemons have to refresh their name server entries periodically and it is inefficient if this causes unnecessary delays in their work. In addition, the ring should have multiple injection points, which means that updates and lookups can be performed using any of the ring members. This is not simple to achieve, although with some additional mechanisms it is possible.

The ring is directed, which means the requests go one way - only acknowledgements are allowed to traverse the ring in the opposite direction. Moreover, one channel is for communication between the servers, and a second one acts as the means by which the clients can communicate with the name servers. Thus, each ring member listens on two ports - on one server requests arrive whilst the clients (i.e. DVM servers, dormant daemons and kernels) requests are transmitted via the second one. These are known as: *server-to-server* and *server-to-client* channels, respectively. It is noteworthy to conclude the description by highlighting that all the communications use the TCP/IP protocol and JDK socket implementation.

It was noted above, that the ring has multiple injection points. Indeed, a client can contact any of the name servers and request an update or a lookup. No matter which ring member has been selected, the client's request should always be processed in the same

[1] The Harness prototype bases its operation on multicast where available. But since multicast is not present on all systems, we assume that a point-to-point protocol is used.

Algorithm 1 Process a client's request.

1. If the request type is 'lookup', return the result from the tables and STOP.
2. If the request type is 'update', do the following:
3. Check if the update is possible considering the contents of the tables.
4. If the answer in step 3 is 'no', send the proper answer to the client and STOP.
5. If the answer is 'yes', do the following:
6. Send the request to the next ring member.
7. Wait for the request to come from the name server prior to you.
8. Send the answer to the client and STOP.

way (in other words, the result of the request must not depend on which ring member is contacted).

Below are the listed requirements, which the ring should satisfy in order to be suitable for use as a name service in Harness:

1. It has to be fault tolerant, both in terms of node crashes and link breaks
2. It has to be fast enough not to cause delays in the work of DVM servers, dormant daemons and kernels
3. All the requests listed in section 2 have to be properly processed and propagated

Unfortunately these principles contradict each other, but as it shown later, it is not possible to satisfy them all. So, in summary:

- the ring consists of up to 8 physical nodes
- the ring is directed
- it has multiple injection points
- it uses the TCP/IP protocol

The general algorithm for handling a client's request is shown in alg. 1 at the top of the page.

According to this algorithm, lookup requests are processed the same way as in the centralized, single-node name service. Only updates are passed around the ring (unless they are impossible to execute according to the local servers tables - in that case it is pointless to send them to the other ring members). Thus, all servers update their tables before the client receives the update acknowledgement.

In order to define an algorithm for processing the incoming requests by the server-to-server channel, we first need to take a closer look at the communication between name servers. Let's suppose a name server NS_i wants to send the request around the ring. It contacts the next available server in the ring NS_{i+1} and sends the request. NS_{i+1} forwards the request to NS_{i+2}, which sends it to NS_{i+3} and so on; finally the request gets back to NS_i.

Initially, NS_i sends the request to NS_{i+1}. NS_{i+1} sends back a 'got ack', which tells NS_i, that NS_{i+1} received the request. At the same time NS_{i+1} forwards the request to NS_{i+2} and waits for it's 'got ack' message. When the 'got ack' from NS_{i+2} is received by NS_{i+1}, NS_{i+1} sends 'fwd ack' to let NS_i know it's request has been successfully forwarded[2]. The algorithm for handling a request coming from server-to-server channel is shown in alg. 2.

[2] In general, when the ring consists of n nodes and we consider sending one message as a time unit, it takes n time units for a request to traverse the ring.

Algorithm 2 Process a server's request.

1. Read the request.
2. If the request originated from this host, send 'got ack' and 'forward ack', perform appropriate actions on the client side and STOP.
3. If someone else is the sender, do the following:
4. Send 'got ack' and send the request to the next server in the ring.
5. Wait for 'got ack' from the next server, and update the local tables according to the request.
6. When it comes, send 'fwd ack' to the previous server.
7. Wait for 'fwd ack' from the next one.
8. When it arrives, consider the request to be forwarded successfully and STOP.

Algorithm 3 Register a DVM (simplified version).

1. Read the request from the client.
2. Check if there is a server for this DVM registered in the local tables.
3. If there is, execute steps 4-10, otherwise goto 11.
4. Check if the server host and port are equal to those from the request.
5. If they are, consider this request to be a refresh of the entry, so reset the timeout and send the request around the ring. When it comes from the other side, send the acknowledgement to the client and STOP.
6. If the answer in 4 is 'no', do the following:
7. Try to ping the current DVM server.
8. If the server is alive, send the information about it to the client and STOP.
9. If the ping failed, send a DPING request (see below) around the ring. Wait until it comes from the other side. If the DPING succeeded, the current DVM server is alive - send appropriate information to the client and STOP.
10. If PING and DPING failed, register the client as a new DVM server, send the request around the ring and when it comes from the other side, send the acknowledgement to the client. Then STOP.
11. Register the client as a new DVM server, send the request around the ring and when it comes back send the acknowledgement to the client. Then STOP.

This is a very general form of the algorithm, details depend on the request type. Some additional actions have to be taken when a server does not receive 'got ack' or 'fwd ack' from the next server in a specified amount of time or when it's not possible to contact the neighbor, those mechanisms are described in section 4.

Step 2 requires further explanation. If a server receives a request and detects it is the sender of the request, then this means that the request traversed the entire ring and was received by all name servers. Therefore it shouldn't be forwarded to the next ring member, because it would cause the message to go around the ring once more. The sender of the message doesn't forward it, but the previous node in the ring still waits for the acknowledgements. If it doesn't receive them, it will assume the server crashed. To avoid that, the sender of the message sends back 'got ack' and 'fwd ack' despite the fact that it did not forward the message.

We now formulate the algorithms for handling different types of client requests. For a request to register a DVM server, the algorithm is shown in alg. 3.

This is a simplified version since under some circumstances there must be some more actions taken. Those special cases are described in section 4. The algorithm for registering a dormant daemon is much simpler and is shown in alg. 4.

Algorithm 4 Register a dormant daemon.

1. Read the request from the client.
2. Put the data into the local tables and reset the timeout.
3. Send the request around the ring. When it comes back, send the acknowledgement to the client and STOP.

As noted before, a list of hosts running dormant daemons is maintained by the name service. Thus, it is unnecessary to check if a particular host has been already registered. If it was, then the entry will be overwritten with the same value.

Recall that lookups do not require an interaction between name servers. This is because all updates are immediately propagated around the ring, therefore all name servers keep exactly the same data and no communication between them is necessary to give an answer to the client.

It is noteworthy at this point to mention the DPING operation, which is specific for the distributed name service and does not appear in the centralized one. When a DPING is requested by one of the ring members, this means that it hasn't been able to contact a DVM server, but that does not necessarily means the DVM server has crashed. The name server asks all the other ring members to ping the DVM server on its behalf. Thus, the algorithm for processing the DPING request is shown in alg. 5.

When the DPING request traverses the ring and finally returns to it's sender, the sender checks the value of the 'pinged' field. If the value is 'true', it means at least one name server was able to contact the DVM server, so the server is alive. If 'pinged' contains 'false', the DVM server is considered to have crashed or is in a minority partition.

Note, that after the first ring member successfully pinged the DVM server, the subsequent name servers do not attempt to contact it again, simply, the request is forwarded. An example of a DPING operation is shown in fig. 2 below.

In this example, NS2 was unable to ping DVMS3, so it requested a DPING. NS3 wasn't able to ping DVMS3 either, so it forwarded the request with 'pinged' set to 'false'. Finally NS4 managed to ping DVMS3, so it set 'pinged' to 'true' in the request. All subsequent name servers forwarded the request, which finally reached NS2.

4 Fault Tolerance In HDNS

The Harness Distributed Name Service is not fully fault-tolerant (although it is much more reliable than a single-node name service). Unfortunately the distribution itself introduces additional types of faults. In the centralized name service, a machine crash

Algorithm 5 DPING.

1. Read the DPING request from server-to-server channel.
2. If the 'pinged' field in the request is set to 'true', then that means one of the previous name servers has managed to contact the DVM server. In this case forward the request without additional actions and STOP.
3. Otherwise, attempt to ping the DVM server.
4. If the ping succeeded, then set the 'pinged' field in the request to 'true' and forward the request. Then STOP.
5. If the ping failed, leave 'false' in the 'pinged' field, forward the request and STOP.

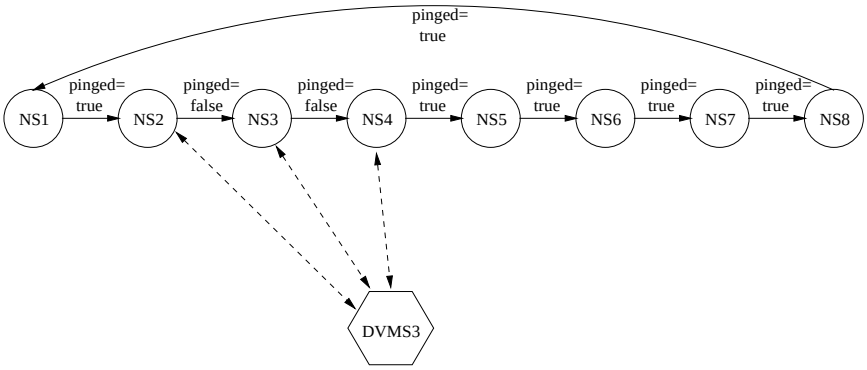

Fig. 2. An Example Of DPING Operation.

or a link failure was the only problems that could cause the whole system to crash. In the distributed version, data can become inconsistent which may be even more dangerous than a node crash. In this section we consider possible scenarios and discuss how HDNS deals with them.

4.1 Message Loss

The conditions in which a request traversing the ring may be lost can be described as: i) if there were no acknowledgements or ii) a single machine crash or a link failure would cause the request to traverse the ring.

Consider that P_{mf} is the probability that a host in the ring crashes exactly at the moment when the message reaches it and before it's able to forward it. P_{lf} will denote the probability that a link between two nodes does not operate. When there are no acknowledgements, the probability that a message is lost equals $P_l = P_{mt} + \frac{1}{n-1}P_{lt}^2$ i.e. if a ring member crashes or becomes isolated when the message reaches it, the message will never make around the ring.

In the version with acknowledgements (as described above), one of the following may take place to cause the message loss:

Recall, that if a name server sending or forwarding a request doesn't receive a 'got ack' or 'fwd ack' message, it assumes that the recipient has crashed and connects to the next server in the ring, circumventing the failed host. To cause message loss, we must visualize a situation in which all acknowledgements have been sent but the message has not been forwarded.

In scenario 1, NS_{i+1} receives the request from NS_i, sends the 'got ack' and forwards the request to NS_{i+2}. After getting 'got ack' from NS_{i+2}, it sends 'fwd ack' to NS_i. From that moment NS_i is convinced the message has been successfully forwarded. Suppose NS_{i+2} sent the 'got ack' but it took an arbitrarily long time to contact the next ring member, so that NS_{i+1} had enough time to send 'fwd ack' while the message is still in NS_{i+2}. Just after NS_{i+1} had sent 'fwd ack', both NS_{i+1} and NS_{i+2} crash (before NS_{i+2} forwarded the message). The only host who could detect the fact that NS_{i+2} hasn't forwarded the message is NS_{i+1}, because it will not get 'fwd ack' from NS_{i+2}. But NS_{i+1} has crashed too, and NS_i got both acknowledgements and will not take care

about the message any longer. In this situation the message is lost and will return to the sender. This situation may occur with the following probability:

$$P_{ldh} = \frac{1}{n-1} P_{mf}^2$$

where P_{mf} is the probability that a single ring member crashes exactly at the moment described in the scenario above.

In scenario 2, the situation is similar, but instead of a host crashing, a link fails. Suppose the connection between the pair NS_{i+1}, NS_{i+2} and all the other ring members stopped operating as soon as NS_{i+1} sent 'fwd ack' to NS_i, while the message is still in NS_{i+2}. In that case the message is 'trapped' in the pair NS_{i+1}, NS_{i+2} and will not traverse further (in fact it will be discarded as soon as NS_{i+2} detects that there is no connection to the message sender and considers the sender to have crashed. The probability of the occurrence of such a situation is:

$$P_{ldl} = \frac{1}{n-1} P_{lf}^2$$

So the model with acknowledgements is as vulnerable to faults, as the model without acknowledgements, but it is at less risk for node crashes. Therefore the probability that a message is lost in HDNS is:

$$P_{ld} = P_{ldh} + P_{ldl} = \frac{1}{n-1}(P_{mf}^2 + P_{lf}^2) < P_l$$

The probability of message loss is much less in the model with acknowledgements, than in the model without them, and notice that the total time necessary for the request to traverse the ring is equal in both cases. Evidently, it requires more messages to be sent, but the request gets back to the sender in the same amount of time in both models.

In case a message has been lost, a thread is created on the sender node for each request sent. If the request doesn't come back in a specified amount of time, the thread re-sends it. So, even if a message is lost as a consequence of two name servers crashing, it will be retransmitted. Thus, to the client it will appear as an increase in processing time.

4.2 Data Inconsistency

As noted above, all the update requests have to be processed in the same manner regardless of which ring members have been contacted by a client. They should also produce consistent answers when they receive lookup requests. The multiple injection points may be an issue when we consider registering DVM servers. Assume two DVM servers attempt to register themselves as servers for the same DVM (let's call it 'Red'). As it was said in section 2, each Harness DVM consists of exactly one DVM server and an arbitrary number of kernels. If both servers attempt to register themselves using the same ring node, there is no problem - the first is registered, and the second one receives information that there is a DVM server for 'Red' already. Suppose then, that they use different nodes to register themselves and there is no DVM server for 'Red' registered, so theoretically both name servers should accept the request according to the contents of their tables. But according to alg. 3, the DVM server being registered gets the acknowledgement from the name server after the request comes back to the name server, and that creates an opportunity to introduce a level of control, thus solving the problem.

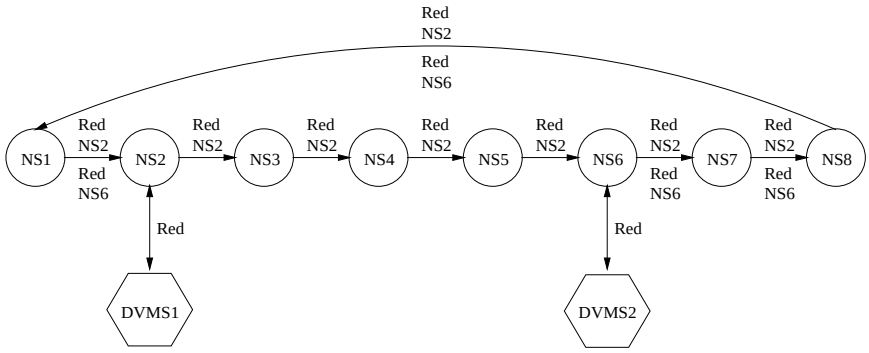

Fig. 3. Priority Based DVM Registration.

As an example, consider the following scenario. NS2 sends it's request around the ring and NS6 will do the same. Both requests contain the same DVM name. It must happen, that the request of NS6 reaches NS2 and the request of NS2 reaches NS6 (not necessarily at the same time, but its certain that the request of NS6 will reach NS2 before it comes back to NS6 and the request of NS2 will reach NS6 before it comes back to NS2). At that moment the server priorities become significant. First consider NS2. It receives the request of NS6 and detects that there is its own request pending for the same DVM. So it compares it's own priority with the priority of the received requests sender (NS6). If it's own priority is higher (it is in our example), it discards the incoming message; i.e. it is not forwarded by NS2 and will never return to NS6. This does not mean that NS6 will never return an answer to DVMS2. However, the request of NS6 will never return to it, NS6 receives the request of NS2 and performs exactly the same procedure. It detects that there is it's own request pending for 'Red' and compares the priorities. The NS6 priority is less than NS2, so NS6 modifies the entry for 'Red' in it's table, forwards the message and instructs DVMS2 that there is another server for 'Red'. This is shown in fig. 3. The request sent by NS2 is above the arrows in this diagram, whilst the request of NS6 is shown below the arrows.

One potential problem in this algorithm is that for a short time, name servers NS7, NS8 and NS1 have invalid entries for 'Red' in their tables and, if asked, will return false answers. This is the time after those servers received the request of NS6 and before they received the request of NS2. To partially avoid this situation, the clients maintain a sorted array of name servers and they first try to contact the server with the highest priority. This does not guarantee that none of the servers with false data will be asked, but it decreases the probability that the situation will arise.

5 Conclusions

A significant effort went into deciding which kind of topology should be used to build the HDNS. All of them have disadvantages and we finally decided to implement the service as a ring of servers with multiple injection points. The working prototype proves the concept. The current Harness release (v 1.7) works with the new name service without any changes in implementation (it just uses one of the ring members). In the next

release (2.0), Harness core classes will be changed so that they will take advantage of the HDNS.

During the testing phase of the prototype there appeared several of the problems described above. After some experimentation it turned out, that in most cases the DNS lookup was causing delays in communication. The host name lookup table solved this problem, while an 'interrupting' thread corrected the issue with the long time of opening a connection when a host is unreachable. After changing the process of opening a connection as described above, the problem no longer occurs.

The new name service, as well as the old one and most of the other Harness services, use object input / output streams to send data. In other words, data being sent has the form of serialized Java objects. The process of serialization and de-serialization in Java takes some time, but fortunately name service do not send large amounts of data, so the serialization does not affect the communication speed.

References

1. M. Migliardi and V. Sunderam. Heterogenous Distributed Virtual Machines In The Harness Metacomputing Framework. In *Proc. of the Eigth Heterogeneous Computing Workshop*, April 1999.
2. M. Migliardi and V. Sunderam. The Harness Metacomputing Framework. In *Proc. of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, March 1999.