IOSR Journal of Computer Engineering (IOSRJCE)
ISSN: 2278-0661 Volume 5, Issue 1 (Sep-Oct. 2012), PP 12-16
ww.iosrjournals.org

A Protocol to Detect and Kill Orphan Processes in Distributed Computer Systems

Shamsudeen.E[1], Dr. V. Sundaram[2]
[1,2] *(Research Scholar, Karpagam University, Coimbatore, India)*

ABSTRACT: *The protocol presented here is to detect and kill orphan processes [1] in distributed computer systems effectively. That is it detects and kills the orphans when they are born. It does not waste any time to kill the orphans as the previous approaches do. All Remote procedure calls (RPC) [21][3][4] are recorded in the global log[5][6] which is associated with the monitor process[5][6] which is kept in the system to watch constantly the RPCs to find out the orphan computations. It is done by the monitor process which sends a token to those clients who participates in the RPCs and find out the active clients and abort [7] or crash processes [7]. The protocol is very much effective in the sense that the orphans are killed immediately after their birth. This protocol also handles the nested invocation [8] of RPCs. So, this protocol maintains data consistency [9] in the system and it avoids the wastage of valuable computer resources. All other approaches wait until the abort or crash process gets rebooted. Till the crash processes get rebooted, the orphans are active and make unwanted result which may lead to inconsistency [10] of data and moreover the wastages of valuable resources like CPU, memory and database etc.*

Keywords: *Remote procedure call, orphan, monitor process*

I. INTRODUCTION

The distributed systems always the problem of orphan computations, the computations whose results is no longer needed. It may happen by aborting the parent process that made the request to the server asking some service or a crash happens to the client who is responsible for the RPC. Many protocols are in the literature to deal with the orphan processes, but all the protocols are active after the rebooting of the parent node. Until then the orphan is allowed to run at the server site. Some protocols do not say anything about the nested invocation.
Our protocol detects and kills the orphan process, the moment they are born. It also deals with the nested transaction. Here we listed three major previous approached which we find in the literature. Let us see the previous common approaches used to detect orphans in distributed systems.

1.1. Nelson's Approach

Nelson [11] discusses orphans and orphan detection in the context of his remote procedure call scheme.

- This app
- roach deals with only the crash orphans
- There is no notion of Nelson's orphans viewing inconsistent data

The first orphan detection scheme nelson proposed is extermination. This scheme delays the recovery from a crash until all orphans created by crash are tracked down and destroyed.

The second approach put forward by Nelson is expiration. In this scheme, each process is assigned a time limit. If a process is still running when its time limits arrives, it is simply destroyed. This scheme sets a time bound an orphan can exist before being destroyed. Unfortunately, the expiration can lead to the destruction of non-orphaned processes.

The final approach is reincarnation. In this scheme, each site maintains a crash count, which he calls an epoch. When a site recovers from crash, it increments its epoch number. The epoch number is piggybacked on every outgoing message from a site. When a site receives a message with a higher epoch number than its own, it destroys all its local processes and increases its own epoch number. Of course, this can result in non-orphan computation being destroyed. To correct this deficiency, Nelson proposes another scheme called gentle reincarnation. It works just like plain reincarnation does except when destroying processes at a site that has just received a higher epoch number on an incoming message. Instead of simply destroying processes, querying up the ancestor chain is done to ascertain if a process is indeed an orphan or not.

Of all the orphan detection schemes presented, Nelson considers the combination expiration and extermination is the best.

1.2. Lampson's Orphan Detection Schemes

As Nelson points out in his thesis, his orphan detection schemes are worked out versions of schemes proposed by Lampson [12]. Lampson also proposes additional scheme to those detailed in Nelson's thesis, deadlining.

Deadlining is an enhancement of expiration, described in the last section. Instead of merely aborting a process when it reaches its time limit, querying is done up the ancestor chain to ascertain if the process is actually an orphan. If the process is not an orphan, its time limit is extended.

1.3. Allchin's Method

Allchin [13] presents a system based on nested atomic actions. His algorithm is separated into two halves-an abort-orphan detection and crash-orphan detection.

Now we present a protocol to detect and kill orphan. It solves the problem of nested orphans; it reduces the time gap between the birth and killing of orphan process because the detection of orphan is done immediately after their birth by a message which is sent by the monitor process that tells the server to kill them off.

II. The Protocol

Here each and every request is logged at global log which is at the monitor process who monitors the entire RPCs of the system. To find out the active and passive clients, the monitor process sends a token to those clients to who made a request to get service from the servers. The token keeps a data structure which will have a status variable associated with all processes who made a request to the servers. The token revolve round the system and return back to the monitor process. The monitor process then finds out the active and abort or crash parent process by checking the status variable, initially the status variable associated with all client processes who made requests will be '0' and if the client is alive, it will set the status variable to '1'.

Immediately after finding out the crash or abort clients, the monitor processes sends messages to kill the computations to the corresponding servers where the orphans are active.

In the protocol we have *monitor end* to indicate the global log and monitor where the all RPCs are recorded. The *client end* indicates the process which made an RPC to get service from the server process and the *server end* is to represent the server where the request of client process is fulfilled.

In the following session we present the pseudo code of the protocol and later we present an instance of the protocol diagrammatically in fig.1.

/begin/

```
Monitor end:
// it sends a token to those processes (client) who made requests to different servers to get services and waiting for the result. are active at client end to find whether parent process are active or not inorder to kill the corresponding orphan process which are active at server end
status variable:=0;
 // initializes status variable
send token ()
client end
receives token;
Case 1:
// client is active and send the token to next node
if client is active
        {
status variable := 1;
                send token ()
}
else
 //client fails and the token do not send to the next node, here fail occurs after receiving the token and before updating it.
        {
status variable:=0;
!send token ()
}
Case-2:
//client is active and after updating the token variable, the node fails, then token should not be transferred to the next node.
if client is active
        {
                status variable:=1;
```

```
                client down;
                !send token ()
        }
else
//client fails and the token do not send to the next node, here fail occurs after receiving the token and before updating it.
        {
status variable:=0;
!send token ()
        }
Case 3:
//client is active, but the link between the client and server fails just after receiving the token and update its status variable to 1, then the token should not be transferred to the next node or monitor.
        if client is active
        {
                status variable:=1
                link fail;
                !send token ()
        }
else
//client fails and the token cannot be  sent to the next node, here fail occurs after receiving the token and before updating it.
        {
                status variable:=0
                !send token ()
        }
!receives the token;
//client is down, and there will be no client to receive the token, so the status variable of that particular process will be '0' and the token will be sent to the next node.
If the client is down
{
!receives the token;
status variable:=0;
}

Monitor end:
//do not receives the token back either link fail, node fail, or abort process, then regenerate the token and send through the network.
!receives token;
        Send token ()
Monitor end;
// successful tour of the token gets back to the monitor and monitor check for the value of the status variable of each processes.
Receive token back.
if status variable ==0
{
status variable:=i[0]
//for process i
send kill (pi,j)
//function to kill process 'i' at server 'j'
Server end:
receives kill message;
kill pi;
}
send token ()
//the new token to be sent with new process details.
/end/
```

Continue this process as long as the global has entries. If global does not have any entry means that there is no RPC is made in the system. Here the complexity of the protocol depends on the number of clients participates in the RPC. If more the number, more time to be taken to evolve round the token through the

system. The token should also contain a time stamp parameter with it. The time stamp is used to set a time out in order to tackle the problem when the token is not coming back after the round through the system.
An instance of the protocol is depicted in the following fig.1. It shows only the RPCs and global log entry.

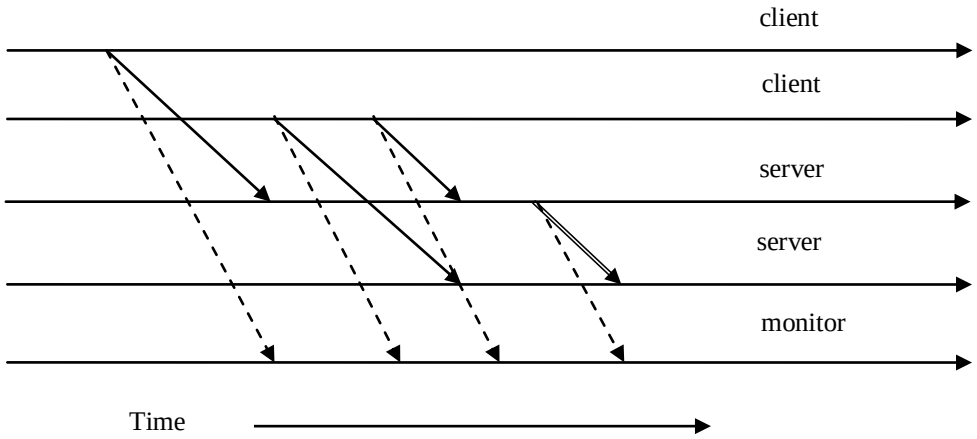

Fig.1. shows how the RPCs and global log entries are made in the global log and monitor protocol**.**

In fig.1,

Arrow indicates an RPC from client to server

It indicates a log entry is being made by the client to the global log at monitor process

It indicates a nested transaction that is being made from the server process to which a client has already made a request

III. BOTTLENECK OF THE PROTOCOL

The problem with this protocol is that if the monitor process fails, then the entire system would not work at all. But, this can be overcome by replicating the monitor process or by select any other process as the monitor process. Any suitable algorithm can be used select the monitor processes among the processes. But the real problem here is none other than the number of RPCs. If more the number, then the performance should be affected. Because the a single tour of the token may take more time. During that period of time some orphan may be active there in the system.
The performance parameters are not only the number of RPCs, but the communication channel capacity, the time taken to set status variable value at the client site.

IV. MERIT AMONG OTHER PROTOCOLS

The all other protocols to handle orphans are active only after the rebooting of the clients from where the request has been made. But, our protocol ensures that the detection and killing of orphan is done immediately after the birth of the orphan computation. It is possible by constantly monitoring the all scenario of RPCs in the system by the monitor process. Since all RPCs are entered in the global log including the nested transaction, this protocol is very efficient to detect and kill all orphans. The previous protocols never say any thing about the orphans generated by nested invocation.

V. CONCLUSION

With this protocol the issues related with the orphan processes are resolved. That is, there is no inconsistency of data occurs nor any wastages of resources either. Because the orphans are killed the time they are born. There is no locking of resources by unwanted computations and hence there is no deadlock [14][15] of resources at all. But the bottleneck with this problem is that if the number of remote procedure call is more, then the time required to revolve round the token through the system is more. The problem associated with this is that during the period of token's tour through the system the orphan may be active at the server site. But still it is far better than other approach because they kill the orphans only after the rebooting of the failed node. So, the protocol is very much effective, when the numbers of RPCs are not more in number.

References

[1] Maurice Herlihy, Nancy Lynch, Michael Merritt, and William Weihl. On the correctness of orphan elimination algorithms., Proc. 17th Annual IEEE Symposium on Fault- Tolerant Computing, July 1987.

[2] Pradeep K. Sinha, Distributed Operating, concepts and design (Prentice Hall India, 2008),36,186,167-230.

[3] Kai Hwang, Advanced Computer Architecture: Parallelism, Scalability, Programmability (McGrawHill International Edition)379-382,627,305-382.

[4] Andrew S. Tenenbaum, Marten Van Steen, Distributed Systems: Principles and Paradigms (Prentice-Hall India, New Delhi 2003), 38,68-85.

[5] Shamsudeen. E, V. Sundaram, An Approach for Orphan Detection, International journal of computer applications, Vol.10.No.5,2010,28-29.

[6] Shamsudeen. E, V. Sundaram, Time Stamp Based Global log and monitor approach to handle orphans in distributed systems, International journal of computer science and network security, Vol 11 No.8, 2011,123-125.

[7] M. Jahanshahi, K. Mostafavi, M.S. Kordafshari, M. Ghlipour, A.T. Haghighat, Two new Approaches for Orphan Detection, Proc. IEEE 19th International Conference on Advanced Information Networking and Applications (ANAI□05), 2005,461-464

[8] S. Pleisch, A. Kupsys, and A. Schiper, Preventing Orphan Requests in the Context of Replicated Invocation. Proc.22nd Symp. on Reliable Distributed Systems, 2003, 119–129.

[9] Gray, J Reuter, A.:Transaction Processing: Concepts and Techniques (Morgan Kaufmann, Sanfrancisco 1993).

[10] Parker, D.S., Jr. Popek, G.J. , Rudisin, G. , Stoughton, A. , Walker, B.J., Walton, E., Chow, J.M. , Edwards, D., Kiser, S. , Kline, C. Detection of Mutual Inconsistency in Distributed Systems, **IEEE Transactions on** Software engineering, SE-9 , Issue: 3 ,May 1983 **,**240-247(inconsistency)

[11] Nelson, Bruce, Remote Procedure Call, Ph.D. thesis, Technical Report CSL-79-3, Xerox Palo Alto Research Center, 1981

[12] B. Lampson,Remote Procedure Calls,Lecture Notes in Computer Science 105.Springer-Verlag, Berlin, 1981, 365-370.

[13] Allchin. J.E, An architecture of reliable decentralized systems. Ph D thesis, Georgia Institute of Technology, Septemeber, 1983

[14] Bipin C. Desai, An Introduction to Database Systems, (Galgotia Publications Pvt. Ltd, New Delhi, 2000),700-705

[15] Ramez Elmasri, shamkant D Navathe, Fundamentals of Database Management Systems (5^{th} Edition, Pearson Educations, New Delhi).