IP Over NBMA Working Group
INTERNET-DRAFT
Expires 21th of April 1997
<draft-mannie-stc-ion-msp-00.txt>

Eric Mannie
Marc De Preter
(ULB-STC)

October 1996

Multicast Synchronization Protocol (MSP)

Status of this Memo

Abstract

This document defines a Multicast Synchronization Protocol (MSP) designed to avoid traditional problems related to the use of unicast (pt-to-pt) synchronization protocols (such as those encountered with OSPF, P-NNI, SCSP, epidemic protocols,...). These protocols imply the establishment and maintenance of a topology of servers (tree, star, line, mesh,...). It is not obvious to find neither the best topology for a given synchronization protocol nor the best algorithm which allows to create this topology. An attempt to study the influence of the spatial distribution (topology) has been made, for instance, in Epidemic algorithms by Xerox Parc, which showed interesting results. Moreover, traditional synchronization protocols notably imply a convergence time and traffic which is proportional to the size of the topology. We believe that reducing the topology to a set of members of a single multicast group could reduce both convergence time and traffic. Note that in that case, no configuration algorithm is required.

1. Introduction

MSP allows to synchronize a replicated database between various servers which are all members of the same server group (SG). It takes advantage of multicast capabilities provided by a growing number of underlying layers. It is suitable in environments supporting multicast (group) addresses (such as ethernet and SMDS) or point-to-multipoint connections (such as ATM). In this context all servers can directly communicate with all servers in the same group, using the underlying multicast capability. No particular topology (except the underlying multicast topology itself) nor configuration algorithm (such as the Spanning Tree,...) are required. No problem due to critical links and topology partitions occurs. This protocol is very robust as an update generated by a server is directly received by all other servers in the same group. Finally, MSP is a generic protocol and is defined independently of the particular database or cache to synchronize.

2. Overview

MSP is a generic protocol defined independently of the particular database to synchronise. The MSP pdus may be either self transported or part of other protocols and used as fields of these protocols. For instance MSP can be supported on the top of an IP multicast service, on the top of an Ethernet network, on the top of an ATM service or it can be a part of Classical IP, MARS or NHRP. It can even be used to synchronize different databases in parallel, in the same pdus.

Each server is the owner of a part of the database (e.g. bindings by local clients) and has a unique server ID. It maintains a copy of the complete database (replicated database), each entry is either locally owned or learned from another server. An entry which belongs to a particular server is tagged with a timestamp (event time) and the server ID of its owner. This timestamp identifies an update of the entry and is set each time the entry is updated. The timestamp is unique in the context of the owner and is the concatenation of a standard time and a sequence number.

All servers are directly connected by a multicast group or a mesh of point-to-multipoint connections. Each entry update generated by a server is sent with its timestamp and the server ID. This update is directly received by all other servers and each local database is updated accordingly. Updates are packed with their timestamps in pdus, which are logically grouped into transactions. Transactions allow to speed up the detection of the loss of a part of the pdus.

Pdus are not positively acknowledged (ACK) by receiving servers. If a server detects some missing pdus, it sends a NACK in the multicast group for the corresponding missing updates identified by their timestamps. In order to avoid an implosion of concurrent NACKs and reduce the total number of transmitted NACKs, a technique similar to IGMP is used. A server waits for a delay randomly chosen between zero and D milliseconds before sending a NACK. If, during this period, another NACK is seen for the same timestamps, the NACK generation is

cancelled and the server waits for the retransmitted updates. This scheme is based on the fact that if a pdu is lost in the context of a multicast group, there are probably more than one server which have not received the pdu. In addition, it should be noted that such pdus are essentially small pdus and that the expected error rate of the underlying layer should be very low (e.g. ATM).

Small hello pdus are generated periodically at each hello interval and include the server id and its last attributed timestamp. Hello pdus are needed to detect the loss of a set of complete transactions.

When a server or a connection (re)starts, either all the database has been lost, or, only the most recent updates have not been received. The corresponding server sends a SYNC pdu to the multicast group, either indicating that all the database has to be retransmitted, or, listing each server ID and its last received timestamp. SYNC are not sent directly; the same scheme as the one used for NACKs is used again; it allows a server to take advantage of the resynchronization requested by another server. Receiving servers send transactions for the missing information, and could use the same scheme as the one used with NACKs and SYNCs. Each server only sends its own updates and only if their timestamps are greater than the requested one. If some timestamps are not available any longer, it means that the corresponding information has been updated by a more recent update and that only this last update has to be retransmitted by the owner. Only more recent updates are retransmitted. Selective SYNCs are resent if a part of the requested updates has not been received, obsolete timestamps are advertised.

For the configuration point of view, only a single multicast address or a list of server addresses has to be configured (e.g. obtained through a "LEC" like configuration server [LANE]). No particular topology has to be built, no configuration algorithm is needed, no problem occurs if a server fails to rebuild the topology !

Finally, if we only consider a topology of servers connected by
point-to-point connections, MSP acts like a traditional
synchronization protocol as explained in the MSP unicast chapter.

```
     S1     S2     S3                    S1  S2
     |      |      |                      \  /\
     |      |      |                       \/  \
   +-----------------+                    S3   S4
   |                 |                    /
   | Multicast Group |---S4              /
   |                 |                  S5-----S6
   +-----------------+                  /
     |      |      |                   /
     |      |      |                  S7
     S5     S6     S7

       Multicast topology   vs. traditional topology
```

3. Server Group

MSP allows to synchronize a replicated database between various
servers which are all members of the same server group (SG). It
ensures that within a short duration, all servers in the SG will have
exactly the same copy of the database. The scope of a server group
could be restricted to servers which are all connected to the same LIS
[Classical], LAG [NHRP], cluster [MARS], ELAN [LANE] or IASG [MPOA].

Each server group (SG) is identified by a server group ID (SGID). This
allows to support multiple server groups at the same time in the same
domain. This SGID only needs to be unique in this domain and could,
for instance, consist of the multicast address used to identify the
servers.

Moreover, each server in a SG is uniquely identified by a server ID
(SID). This ID could be the internetwork layer address of the server
itself, e.g. its IP address. Each pdu transmitted by a server will be
tagged with its server group ID (SGID) and server ID (SID). Using a
complete internetwork layer address as SID allows to quickly identify
a server and to facilitate the management. Both the SGID and the SID
are represented on 32 bits.

4. Underlying Layer

MSP is designed to take advantage of multicast capabilities provided
by the underlying layer. This version mainly focuses on the use of
unidirectional point-to-multipoint connections and full multicast

service such as supported by an Ethernet network. MSP may be supported over the IP layer itself or over any NBMA network such as an SMDS or ATM (UNI3.0/3.1 or later) network.

It should be noted that, if no multicast capability at all is supported by the underlying layer, MSP mainly acts like a traditional point-to-point synchronization protocol (see specific chapter).

5. Topology

The topology is reduced to the scope of a single multicast group. No particular algorithm nor protocol are required to establish and maintain this topology. The address of the multicast group or the list of all members may be directly obtained through a configuration server as the LECS [LANE].

All servers are directly connected by a multicast group or a mesh of point-to-multipoint connections. In the worst case, n point-to-multipoint connections are needed when n servers have to be synchronized. In a near future, when multipoint- to-multipoint connections will be available, the need to support n connections will be cancelled.

It is important to note that these point-to-multipoint connections are only supported by servers (never by clients) and that these servers could be efficiently implemented over internetworking units, such as ATM switches. In addition, the behaviour of these servers is very static as they do not appear and disappear continuously in a server group. This considerably simplifies the management of the connections.

A multicast topology is very robust as every message is directly received by all servers in the SG and as there is no single point of failure or forwarding point at the server level. The complexity of establishing the topology and dealing with dynamic topology partitioning is left to the underlying layer, at the level of routers or switches.

Resources required by servers for message generation are reduced since a message is once sent and is never repeated from server to server. Forwarding of messages is better achieved by transport protocols than by synchronization protocols. It also results that the time needed by an update to reach all servers is very reduced and not directly proportional to the number of servers.

More resources are required to receive messages but this process may be optimized by using appropriate filtering on incoming messages and database lookup, e.g. using dedicated hardware such as CAM (contents access memory). The same kind of problem is solved in Ethernet networks. Filtering on an incoming UPDATE pdu received from a given server is easily applied, by comparing the recorded last received timestamp from that server with the smallest (first) or largest (last) timestamp of that pdu. If the first timestamp in the pdu is greater than the recorded timestamp, the complete pdu updates the local database. If the last timestamp in the pdu is smaller than the recorded timestamp, the complete pdu may be dropped (except if in response to a NACK). Otherwise, all entries between the recorded timestamp and the end of the UPDATE pdu update the local database.

6. Database and Timestamp

MSP is defined independently of the particular database (cache) to synchronise. Database entries are transparently transported and are defined by the particular protocols whose databases are synchronized. The encoding of these entries will be defined in specific appendixes of this document or in companion documents. The database entries describe for instance bindings between IP addresses and ATM addresses. Unlike other synchronization protocols, MSP does not use summaries of database entries. MSP identifies events and ensures that the most recent event for each entry has been received by all servers.

Each server is the owner of the part of the database which is generated by its own clients. It maintains a copy of the complete database but is only responsible for the entries it owns and can only transmit these entries. For instance, each entry in a database may be either tagged as locally owned or learned from another server.

Each event (client action) in the database is identified by a timestamp which is unique in the context of the server which generates that event. This timestamp is the concatenation of a standard time (e.g. GMT) and a real contiguous sequence number. This time may be local to each server, a global time is not needed by this protocol, i.e. clocks do not need to be synchronized. The sequence number is incremented by one at each new event. It allows also to support more than one event per tick of the standard time.

Each entry in the database is associated with a timestamp set by the owner server at the moment when the client generates or updates the entry. A server cannot change timestamps for entries it does not own. Timestamps are used by servers to identify missing information in their database.

If a server receives an entry which is already included in its database, it must compare the two timestamps and keep the entry with the most recent (greater) timestamp.

A timestamp X is greater (more recent) than a timestamp Y if the standard time of X is greater than the standard time of Y or if they are equal, the sequence number of X is greater than the sequence number of Y.

When a server starts for the first time or looses its knowledge of its current timestamp (after a crash for instance), it only has to generate a new value for the standard time and restart the sequence numbering at zero. The same scheme is applied when the sequence number wraps out. The standard time is never modified when a timestamp sequence number is incremented. It is only used to insure a unique value during the lifetime of a server.

7. Transactions

Database entries are transmitted in variable length UPDATE pdus. The size of an UPDATE pdu may be limited in order to fit in the particular MTU supported by the underlying layer. UPDATE pdus related to events close in time are logically grouped into transactions. A transaction is delimited by a start flag in its first UPDATE pdu and by a stop flag in its last UPDATE pdu. The shortest transaction contains a single UPDATE pdu with both the start and stop flags set.

```
     +------------+-------------------+
     | Pdu header | Entry 1...        |
     |(#seq,...)  |        ...Entry n |
     +------------+-------------------+

               UPDATE pdu
```

Transactions are generated in response to synchronization request (SYNC received) or in response to a NACK or when a set of entries have to be flooded. An empty transaction is sent to indicate that the requested timestamps are not available any more (obsolete), following a synchronization request or a NACK. Such a transaction is made of one empty UPDATE pdu identifying the requested timestamps.

Transactions are not numbered but all UPDATE pdus are numbered sequentially among all transactions generated by the same server. All entries in a transaction and in its subsequent UPDATE pdus are sorted by timestamp order. No sort algorithm is needed at all, only a sorted list per server has to be maintained (trivial since timestamps are generated in order). The numbering of UPDATE pdus is contiguous so that a server can immediately detect missing pdus. UPDATE pdu sequence numbers should not be confused with timestamp sequence numbers. PDU sequence numbers are needed since timestamp sequence numbers are not always contiguous (if a server [re-]starts or when entries are obsoleted).

```
     +-----
     |
     |   +------------+-------------------+
     |   | Pdu header | Entry 1...        | UPDATE
     |   |(#seq,...)  |        ...Entry n |   pdu
     |   +------------+-------------------+
     |              .
     |              .
     |              .
     |   +------------+-------------------+
     |   | Pdu header | Entry 1...        | UPDATE
     |   |(#seq,...)  |        ...Entry n |   pdu
     |   +------------+-------------------+
     +-----

            A transaction
```

Transactions are useful in order to detect faster the loss of the last UPDATE pdus sent by a server. This loss may be detected either by a timer waiting for the end of the current transaction, or when the first UPDATE pdu of the next transaction is received or when the next HELLO pdu is received from the server generating that transaction. These HELLO pdus are generated regularly by each server to indicate its last timestamp value.

A transaction starts either immediately when its first UPDATE pdu is built or after a small random delay in order to avoid multicast storms. During this delay, new updates may be generated by clients and added in the transaction.

If a gap in the numbering of UPDATE pdus is detected, it means either that a part of a transaction has been lost or that a complete transaction or set of transactions have been lost. A NACK pdu is sent, indicating the last received timestamp before the gap and the first received timestamp just after the gap. The last received timestamp before the gap was received in an UPDATE pdu in the same transaction or in a previous transaction.

If after a timeout (NACK interval), the requested entries have not been retransmitted (in a new transaction), a NACK pdu is retransmitted for the same timestamps. After a maximum number of NACK retransmission, the corresponding server is considered as not available any more. All its updates are kept physically in the database but the server memorizes the fact that the last valid received timestamp was the last one received before the gap. When the previously unreachable server will become available again, it will sent an HELLO or a SYNC or a transaction and the current server will discover that a set of timestamps are missing. A NACK pdu consist in a list of server IDs with a list of timestamp intervals for each of these IDs. In the simplest case, a NACK pdu is made of a single ID with a single interval. A specific timestamp value is used to indicate an open interval such as [x, [(i.e. from value x to the infinite).

If a server crashes and loses its current UPDATE pdu sequence number, it restarts the numbering at zero. In that case no problem occurs as it resynchronizes its database as described in the synchronization chapter. In order to be able to detect a gap in UPDATE streams, a server keeps the last pdu sequence number received from each server. After having received a SYNC pdu, a server must transparently set that number to the value contained in the next received UPDATE pdu from the corresponding server, without checking for a gap.

MSP may be implemented over a service which does not prevent pdu misordering. In that case, a server should wait for a small timer before deciding that a pdu is lost, in order to have a chance to re-order the pdus. When that timer expires, a gap is detected and a NACK is sent.

8. Synchronization

The synchronization of information is required in two cases: when an update has to be flooded and when a server has to build or rebuild its database. The first case is detailed in the Update flooding chapter. The last case is the synchronization process itself. It occurs, for instance, when a server joins the SG for the first time without having any idea of any existing binding or when a server joins the SG already having a part of the bindings (e.g. when a broken underlying connection is rebuilt).

A server having to synchronize with the rest of the group will first build a list made of all server IDs included in its database (possibly empty). For each entry in the list, it adds the corresponding highest known timestamp. This list is inserted in a SYNC pdu, together with

the total number of elements, the synchronizing server ID and its own highest timestamp. This pdu is multicasted to the group.

Each server receiving a SYNC pdu scans the list for its own ID. If it finds its ID, it builds a transaction containing all entries locally owned and whose timestamps are greater than the required one. If that required timestamp is greater or equal to its own highest timestamp, no entries have to be send and an empty transaction is build, signalling that the synchronizing server is up to date. Finally, the transaction is multicasted to the group.

If a server does not find its ID in the list included in a SYNC pdu, it means either that it has joined the group while the synchronizing server was unreachable or that it does not own any entry in the global database. In the first case, it has to build and send a transaction for all entries it owns. In the last case, it does not have to send any transaction.

Finally, each server receiving a SYNC pdu checks the synchronizing server timestamp. If the local timestamp associated with the synchronizing server is lower than the received one, the server has not received a part of the synchronizing server own database and sends a NACK.

The SYNC pdu is a kind of summary of the database known by the synchronizing server. Its length is bounded by the total number of servers in the SG. If a SYNC pdu does not reach a subset of the servers, the synchronizing server will not receive any transaction in response from these servers and will retransmit, after a timeout, a SYNC pdu for these servers only.

The multicasting of each transaction improves the robustness of the protocol and allows also other servers to learn entries before starting their own synchronization (if still needed after the silent listening). Transactions in response to a SYNC may be multicasted on a separate multicast address on which only servers which have to synchronize are listening. This last solution reduces the traffic which is globally multicasted and allows also each server to independently decide if it wants or not to receive the synchronization traffic.

If all servers or a large number of servers have lost their connectivity at the same time, the multicast scheme is very efficient. If the real multicasting is not supported and if the transmission on point-to-multipoint connections is not desired, it is possible to use an on-demand point-to-point connections.

9. Update Flooding

Each time a client modifies its entry in a server, a new update is generated and has to be flooded to all servers in the SG. The owner server associates a timestamp with the update and builds a transaction to flood that update. This update is included in an UPDATE pdu. The

server may send a transaction for that single update or it may group a number of updates together in one or more UPDATE pdus in the same transaction. Transactions may be sent immediately or after a small random delay (see chapter on multicast storm).

10. Hello Pdu

Small HELLO pdus are sent periodically at each hello interval. Each pdu includes the server ID and the last used timestamp of the sender. This timestamp allows to detect the loss of a part of the updates sent by the server which has generated the HELLO pdu. In particular, it allows to detect the loss of a complete transaction or a set of complete transactions.

When a server receives a given number of HELLO pdus indicating that it has missed a few updates (its timestamps from the sending servers are out of date), it may decide to resynchronize its database and generate a SYNC. Otherwise, it may send an NACK pdu for each of these servers.

11. Multicast Storm

In order to avoid a multicast storm of NACKs when some UPDATE pdus are lost or a storm of SYNCs when many servers have to synchronize at the same time or a storm of transaction, a technique similar to IGMP may be used. Before sending a NACK, a SYNC or the beginning of a transaction, a server may wait for a small random delay between 0 and D milliseconds. During this delay, the server listens to MSP pdus, receives all transactions and updates its database. This silent listening may result in a decreasing traffic and cancel some local operations as explained hereafter.

If a server wants to send a NACK and that during the random delay, a NACK is seen for the same set or subset of timestamps, the server waits for the responding transactions. If after, that delay all of its requested timestamps have been received, the generation of the NACK is cancelled. The previous explanations on NACK retransmission are also applicable here.

If a server wants to send a SYNC and that during the random delay, other compatible SYNCs have been seen, it waits for the corresponding transactions and decides after if its SYNC is still needed.

If a server wants to send a transaction in response to a SYNC, it may also wait for a random delay in order to limit the number of simultaneous transactions transmitted and/or received, to decrease the amount of resources needed.

12. Unicast MSP

MSP is a multicast or point-to-multipoint protocol but may be also used in an unicast or point-to-point environment. In that case it acts like a traditional synchronization protocol, except mainly that each UPDATE pdu doesn't need to be acked one by one, and that after a

failure no complete database summary has to be exchanged in two ways each time. In this last case, only two small SYNC pdus are exchanged and each server acts as a proxy for the information owned by other servers behind him. Random delays are not needed any more, since there is only one sender for each direction.

Of course, as we are back in the point-to-point case, an algorithm is again needed to establish and maintain the topology. Each server knows automatically the servers for which it must act as a proxy by listening the Hello pdus and learning the position of each server. A proxy server generates or forwards Hello pdus for servers it represents.

A server wishing to synchronize will send a SYNC pdu to each server it is connected to. These servers will respond with their own updates and those of the servers they represent. The SYNC scheme is the same as the one used in the multicast case, except for the random delay technique.

A server knows which servers it represents, by keeping a trace of the connections where HELLO pdus are received from. It represents all the servers that sends HELLO pdus on all its connections other than the one where it has received the SYNC pdu.

When a transaction is received from a neighbour server, the receiving server must directly repeat this update on all its connections, except the one on which it was received.

A server must also respond to each received NACK destinated to itself or to one of the servers for which it acts as a proxy. In addition, it does not need to wait for a random delay when it generates a NACK.

13. Further Study (not in this version)

In order to reduce the number of pdu formats, a SYNC pdu could be implemented as a NACK pdu. In that case, a flag is used to indicate that the NACK is a synchronization. Only the last received timestamp (open interval) is given for each known server. A server which does not see its ID in that list must retransmit all its own entries.

Transactions could be suppressed if the HELLO pdu rate is high enough to allow to quickly detect the loss of the last transmitted pdu from a given server. In the current version, the reception of the last pdu of a transaction allows to know that a bundle of pdus have been transmitted and that no further pdu must be waited before the beginning of a next transaction.

14. Security

When MSP is embedded in another protocol, security considerations are mainly covered by this specific protocol. Detailed security analysis of this protocol is for further study.

Conclusion

MSP is a generic multicast synchronization protocol which may also act as traditional unicast protocol. It reduces the traffic by identifying events in the database in place of using database summaries, and by supporting negative acknowledgments (NACK) in place of systematic ACKs. It is particularly suitable in environments with a low error rate such as ATM. It reduces the convergence time and improves the robustness by using a multicast topology where updates are directly received by all servers. No single point of failure exists. No configuration algorithm and protocol are needed, no specific problem occurs if the topology partitions. It takes advantage of the fact that the forwarding of information and the dynamic routing is better achieved by well-known dedicated protocols such as interworking and routing protocols which are implemented anyway to support the normal data transfer service.

References

[MARS] "Support for Multicast over UNI 3.0/3.1 based ATM Networks.", Armitage, draft-ietf-ipatm-ipmc-12.txt.

[NHRP] "NBMA Next Hop Resolution Protocol (NHRP)", Luciani, Katz, Piscitello, Cole, draft-ietf-rolc-nhrp-09.txt.

[MPOA] "Baseline Text for MPOA, draft", C. Brown, ATM Forum 95-0824R6, February 1996.

[Classical] "Classical IP and ARP over ATM", Laubach, RFC 1577.

[SCSP] "Server Cache Synchronization Protocol (SCSP) - NBMA", J. Luciani et al., draft-luciani-rolc-scsp-03.txt

[Epidemic] "Epidemic Algorithms for Replicated Database Maintenance", Demers et al., Xerox PARC.

[LANE] LAN Emulation over ATM Version 1.0, ATM Forum af-lane-0021.000, January 1995.

[LNNI] LAN Emulation over ATM Version 2 - LNNI specification - Draft 3 ATM Forum 95-1082R3, April 1996.

[IGMP] "Host Extensions for IP Multicasting", S. Deering, STD 5, rfc1112, Stanford University, February 1989.

[OSPF] "OSPF Version 2", Moy, RFC1583.

[PNNI] "PNNI Specification version 1", Dykeman, Goguen, ATM Forum af-pnni-055.000, March 1996.

Acknowledgments

Thanks to all who have contributed but particular thanks to Andy Malis from Nexen and Ramin Najmabadi Kia from ULB.

Author's Address

Eric Mannie
Brussels University (ULB)
Service Telematique et Communication
CP 230, bld du Triomphe
1050 Brussels, Belgium
phone: +32-2-650.57.17
fax: +32-2-629.38.16
email: mannie@helios.iihe.ac.be

Marc De Preter
Brussels University (ULB)
Service Telematique et Communication
CP 230, bld du Triomphe
1050 Brussels, Belgium
phone: +32-2-650.57.17
fax: +32-2-629.38.16
email: depreter@helios.iihe.ac.be

Appendix 1 - PDU Format

For further study