A General Framework for Highly Available Services Based on Group Communication

Alan Fekete* Idit Keidar†

November 1, 2000

Abstract

We present a general framework for building highly available services. The framework uses group communication to coordinate a collection of servers. Our framework is configurable, in that one can adjust parameters such as the number of servers and the extent to which they are synchronized. We analyze the scenarios that can lead to the service availability being temporarily compromised, and we discuss the tradeoffs that govern the choice of parameters.

*Department of Computer Science F09 University of Sydney 2006, Australia. E-mail: fekete@cs.su.oz.au

†MIT Lab for Computer Science, 545 Technology Square, Cambridge, MA, 02139, USA. Phone: (617) 253 6054. E-mail: idish@theory.lcs.mit.edu

1 Introduction

We present a general framework for building a family of highly available services. The services we consider are stateful: clients interact with the service in *sessions*; throughout a session, the service stores changing *context* information for the client. For example, in a *video-on-demand (VoD)* service [2], the changing context for a session includes the movie a client is watching and the client's current location in that movie. In this paper we do not address updates to the content stored at the servers, for example, the movies in a VoD service; we assume a separate mechanism for these.

We use replication to achieve availability: A service is provided by a collection of servers. The set of servers may change dynamically due to failures and also when new servers are brought up to alleviate the load on existing ones. A client may be migrated from one server to another during an on-going session; the client is unaware of changes in the service provider.

Although the service is designed to be highly available, certain failure patterns can lead to undesirable behaviors such as temporary loss of service. The risk for such undesirable events can be minimized at the cost of additional resources: increasing the number of servers and the level of synchronization among them. Each individual highly available service may have a different *policy* for balancing these risks and costs. Our framework provides the *mechanism* for implementing various different policies by allowing a service builder to configure a number of parameters.

An important contribution of this paper is the risk analysis we offer for the suggested framework. We find what patterns of faults risk leaving the service not available to a client. We examine how the likelihood of such patterns can be reduced by carefully adjusting some parameters in the framework, and also the cost tradeoffs implied in such adjustments. We concentrate on understanding the issues that should guide the setting of the parameters; once a policy is chosen, its enforcement could be automated through techniques such as spawning new servers when needed, as described in [5], but we do not deal with that in this paper.

Our framework exploits a *virtually synchronous group communication system (GCS)* [3, 1, 7] as a building block. Our starting point for this work was the VoD service of [2], which exploits group communication in order to have the service remain available through node and link breakdowns, including network partitions. The VoD service uses a number of different groups at three different scales, where some groups are monitoring and controlling the membership of others. The success of this approach is evident in the small size of the code: the VoD server is written in under 2500 lines of C++ code, which provides all the fault-tolerance logic as well as managing the access and transmission of the movies.

In this paper, we generalize the specific design of [2] to give an architectural framework for a class of highly available services. The service of [2] is only one instance of our framework; the framework allows a wide range of services to be implemented and a wide range of fault-tolerance parameters to be configured for implementing different availability policies.

2 Design Goals

The type of service that we envision is one where many clients concurrently access the service, but each individual client does not need to use the service all the time. For each client its use of the service is divided into sessions; the client is connected to the service for the duration of a session, and then it disconnects until a later time when it begins a new session. Within a session, the service will send the client information it requests, in the form of messages called *responses*. We do not assume that within a session the interactions follow a protocol of precisely paired request and response; it is also possible that a request from the client leads to a stream of responses.

The state of the service is divided into two separate aspects: there is a large amount of information, the *content*, that is relevant to multiple clients. Each response the client receives is part of, or derived from part of, the content. Also, there is some state information which concerns a particular session. This *session context* determines which parts of the content the client wants to receive in responses, and how those responses should be sent. The session context can be altered as a result of requests from the client, and it can also change to reflect the fact that certain responses have been sent to the client.

We will focus on services where changes to the content are infrequent, and where there are not strong consistency requirements on when clients notice the changes. Thus in this paper we will not deal at all with changes to the content, supposing that they happen outside the framework we are describing. However, changes to the session context happen frequently. We also suppose that the content is composed from a number of separate *content units*, and that each session involves access to one content unit only.

The VoD service discussed above provides one example that fits our domain of interest. Here each movie is a separate content unit. A session involves a client watching one movie. The movie is represented by a sequence of frames; each frame is sent in a message as one response. The session context includes indication of the point within the movie where the client is watching; this can be changed by control messages from the client (e.g., "skip to the start of scene 4"), but the location also advances as each frame is sent to the client. The session context also includes information on the rate at which the client wants to receive frames, etc.

A distance-education service has shared state which has many "learning objects" including lecture notes, animations, quiz questions etc; these are grouped into topics. One topic is a content unit for the service. A session involves a client ("student") studying a topic, by downloading and interacting with some of the relevant learning objects. The learning objects to be viewed are chosen dynamically during the lesson, based on both the student's wishes (e.g., following hyper-links between objects) and on the student's performance on quiz questions (e.g., the service may provide more detailed explanations if the last quiz grade is low).

A third example is a search service which allows a client to make successively narrower queries by restricting the search in one query to within the result set of earlier ones. A possible query would be "select from the results of query 3 where also publication date is after 1995" or "find the intersection of the results of query 4 with query 7"; in general, the session context is the list of previous result sets.

We will provide a framework for implementing any service that fits the pattern described above, with unchanging content and changing session-specific context. The basic design goal for our framework is that the service should be available, that is, the service should provide the responses that clients want. The service should be able to overcome process and network failures, and should be able to serve a variable number of clients. The availability requirements lead to a design where the service is provided by replicated servers. We therefore assume that each content unit can be served by several servers, but we do not require that every server provide every content unit of the whole service. Thus, the replication is partial, not total.

A second important design goal is to make the service as flexible as possible, and at the same time to keep the client design as simple as possible. For example, the service should have the flexibility to allow for dynamic changes in the set of service providers; the client should not be aware of such changes. Therefore, achieving availability should not be the client's responsibility.

When a client makes a request, it should get its response from one of the servers. It is natural to try to keep the same server throughout a single session, but this may not always be possible: the server may crash or may be overloaded. Therefore, it is clear that in some situations the client may need to be migrated to another server during an on-going session. As explained above, such migrations should be initiated and managed by the service, not by the client.

Let us examine what potential problems can arise when a server fails and the client is migrated to another one. First, a request may be lost, in which case a corresponding response will fail to arrive. Next, assume that a response does arrive. Note that because we have treated the content as static, each response contains a correct subset of the content (i.e., a response can never be incorrect). It may, however, be a duplicate. Also, an unwanted response may arrive because the service has been sending responses based on out-of-date context (e.g., a VoD service may have lost the context update where a client asked to jump to a new location, and then continue to send frames from the previous location).

We can therefore see the following availability design goals:

- First, there ought to be exactly one server at a time that is sending responses for a particular session.
- Also, the server that is responding should have a session context that is up-to-date, reflecting all requests from the client during this session and all previous responses.

3 The Solution

We suggest a framework for highly available services. In our framework, a service is provided by a collection of servers, each capable of serving some of the content units of the service, but not necessarily all of them.

The set of servers may change dynamically due to failures and also when new servers are brought up to alleviate the load on existing ones. Clients using the service are generally unaware of such changes.

The framework provides the *mechanism* for meeting the availability design goals of the previous section under a variety of circumstances. When instantiating the framework to build an actual service, one has to define the availability *policy*; that is, to what extent would the design goals be met under different circumstances, and at what expense. We therefore present the framework with several *configurable* parameters. In the next section, we study the tradeoffs involved in different choices for parameter values.

3.1 Meeting the design goals

Let us examine the design goals of the previous section. First, at a given time, we try to have a single server serve each client session in-progress. We call this server the *primary* server for the session. There can be a single primary server when there are available servers that can communicate with the client, and when the network is stable enough to allow these servers to agree among them which one of them will be primary[1].

The primary server of an on-going session may have to change, either due to a crash, or preemptively for load balancing purposes. If the server crashes in the midst of a session, client context information may be lost. Information loss may lead to loss of service, or to missing, duplicate, or irrelevant responses. Replicating context information may minimize loss, but may also be costly.

Consider, for example, the VoD service. If the primary server crashes in the midst of sending a video stream to a client, a new primary server will take-over and serve the client. To this end, the new primary server needs to know of the session's existence. In order to send the client the correct video frames, the server also needs to know which frames the previous primary had sent before crashing. It could know the exact location in the stream where the server had failed by listening to all the communication between the primary and the client. However, since the video stream has a high bandwidth, this would result in significant load. Instead, the primary can periodically update other servers about its location in the movie. This way, these servers' client context information would not perfectly up-to-date, but also not too far off. In the VoD service of [2], such updates are sent every half a second. Thus, upon migration, a new primary may send half a second of duplicate video frames to the client and the server may be unaware of context updates (e.g., requests to skip to a different part of the movie) sent by the client in the last half a second.

In general, there is a tradeoff between the cost of keeping up-to-date context information, and the improved availability such information allows for. To balance these parameters, our framework keeps context information with three levels of freshness. The primary server of a session always has the most up-to-date context information for the session, reflecting exactly the responses which were sent by the primary to the client, and all the context updates received from the client. The primary server periodically propagates context updates to a group of servers providing the same content unit. These servers maintain a replicated data structure called the *unit database*. The unit database keeps track of the sessions that exist for a particular content unit, the allocation of servers to these sessions, and session context information as periodically propagated by each primary. We use properties of GCS to ensure that the unit databases are consistent. The number of servers which contain replicas of the content, and the period between propagation messages, are both configurable. The freshness of the context information in the unit database is mainly determined by the frequency of the periodic updates.

At an intermediate scale, we introduce the notion of *backup* servers. Any number of backup servers per session are chosen among the servers that have replicas of the content unit. In addition to the periodic updates from the primary, the backup servers listen to context update messages from the client, but not to the responses of the primary. This mechanism eliminates the risk of losing client requests upon migration to a backup, but not the risk of sending duplicate responses. In a setting, like VoD, where client requests are fewer and smaller than server responses, this policy does not significantly load the backup servers. Our design will use properties of GCS to guarantee that client context updates are at least as current as information in the unit database. The number of backup servers per session is configurable.

[1]If the network is asynchronous, then it can prevent such agreement [4]. However, while the network is fairly stable, and process failures can be consistently detected, such agreement can be reached.

3.2 Using group communication

The solution exploits a partitionable virtually synchronous GCS as a building block. The GCS includes a membership service, which provides each server with a *view* of the network topology. If a process is a member of several groups, its failure or separation from the others is reflected consistently in new views for these groups. At times when the network situation is stable, views are precise (see [7]). The GCS also carries *multicast* messages addressed to groups; it supports reliable delivery, totally ordered in each group, with causal order preserved across groups. Delivery is *virtually synchronous*, that is, when members move together from one view to another, they all receive the same messages in the earlier view. The GCS supports *open groups*, that is, a process does not need to be a member of a multicast group in order to send a message to that group.

The service creates three kinds of multicast groups:

Service group consists of all the servers. This group serves as a point of contact for clients to connect to the service. We assume that all clients have a priori knowledge of this group's name.

Content group (one for each content unit) consists of those servers that store a replica of the specific content unit, for example, the servers that hold a specific movie in the VoD service. All content groups are subsets of the service group, and these groups may overlap.

Session group (one for each currently connected session) a subset of the content group consisting of the primary server and a number of backup servers.

The set of servers participating in each of these groups may change at any time. The service and content groups may change due to server failures and also as new servers are brought up to alleviate the load on existing servers. A session group may change, either due to a server crash, or for load balancing purposes. A client is not aware of such changes, as it uses use the abstract group to communicate with whichever servers are currently in this group. This group layout generalizes the approach of [2], where similar groups are created, but with session groups consisting of a single server – that is, there are no backup servers.

3.3 Client interactions with groups

When a client wishes to use the service, it sends a message to the service group. When this message is received, the servers send to the client the list of available content units, and the content group name for each of them. The client chooses a content group from this list, and sends a **start-session** message to it.

In response to the **start-session** request, one of the servers in the content group selects itself to be the primary server for this client, and a number of other servers select themselves to be backup servers. We discuss the selection process below. The selected servers (primary and backups) join a new group, which will be the session group for this client. The group name is computed deterministically by each of the servers. The primary server then notifies the client of the session group name.

Once the session has started, the client does not deal with either the service group or the content group. The client sends all of its requests to the session group. Only the primary server sends responses to the client, and these are sent in point-to-point messages.

3.4 Managing the groups

When a **start-session** message from a new client is received in the content group, each server that receives it adds the client to the unit database, and applies a deterministic function to the unit database in order to select lightly-loaded primary and backup servers for this client. Thanks to total message ordering, the function is evaluated over identical databases at the different servers, and all the servers choose the same primary and backup servers. The selected servers join the session group.

Whenever the membership of the content group changes as a result of a server crash or join, the members receive a new view from the GCS. Upon receiving the new view, the servers evenly re-distribute the clients among them.

If the content group membership change notification reflects server failures only, then virtual synchrony semantics allow the servers to immediately reach a consistent decision as to which clients each server will serve *without* exchanging additional information; virtual synchrony guarantees that all the servers have received the same set of messages before the membership change notification (see [7]), thus, all the servers in the group have identical unit databases at the point when they get the view. Each surviving server in the content group applies a deterministic function to the unit database in order to select primary and backup servers for the clients of the failed servers. The function is chosen so that the new primary assigned will be the former primary if possible, or one of the former backups, if the former primary has failed but some former backup remains in the group. The ability to re-distribute the clients immediately without first exchanging messages allows servers to quickly take over failed servers' clients.

If a content group change reflects the joining of new servers (and possibly failures as well), then all the servers first exchange information about clients, and then use the exchanged information to decide which clients each of them will serve. The allocation is done deterministically based on the combined information, in such a way as to balance the load fairly. For migrated clients, the old primary sends up-to-date context information to the new primary.

Changes in the session group membership are dealt with as follows: First, any new primary and backup servers that were not previously in the session group join it. Then the members that should leave the session group do so. Now the primary server begins sending responses to the client, and also it begins propagating the session status at the appropriate times.

4 Analysis of Fault-Tolerance

We now examine the framework that was presented in Section 3, to see how well it meets the design goals articulated in Section 2. In particular, we want to see which failure patterns might lead to clients which not getting the responses they want. We will examine the tradeoffs involved in different settings of the configurable aspects of the system framework.

The first design goal is that a given client should receive information from exactly one server at any time. As explained in Section 3, the group communication service ensures that, in times of stability in the underlying communication layer, all members of a session group have the same information about the group membership; thanks to the total order and virtual synchrony, all have identical unit databases. Thus, when the members independently apply the deterministic function to decide which member will be the primary server for the session, exactly one member will elect itself as the primary, and respond to the client. Thus the scenarios which can lead to a client not having a unique primary server are the following:

- The group communication membership service might give different views of the membership to different servers, during periods when a view change has begun but is not completing properly. This can only occur while the underlying transmission system is not stable.
- Every server which can provide this content may have either crashed or disconnected from the client. Clearly availability is impossible in a scenario such as this. The probability of this scenario can be reduced by increasing the degree of replication.
- The session group may have partitioned, with at least two partitions each seeing the given client as connected to it. This can only happen while the underlying transmission system is not transitive: that is there are servers which can't communicate with one another, but can both communicate with the client. This is very unlikely in a LAN environment, but it does occur sometimes in WANs.

The second important design goal for an available service is that the primary server have an up-to-date context. The context depends on both the messages sent by the client, and on knowledge of which responses have been sent to the client. We investigate these aspects separately, since they have different impacts. If a primary has missed a context update from the client, then it may send responses that are completely unrelated to the clients current wishes. On the other hand, ignorance about which responses have been sent is less serious, leading at worst to duplicate responses.

A context message sent by the client may be not known to its current primary in case the message was sent before this primary was a member of the session group, and the information in it was not yet propagated to the content group. For this to happen, all the previous members of the session group must have failed (or disconnected from the client) either before receiving the context message or before propagating it to the content group.

As for server responses, there can be uncertainly about those responses that might have been sent in the interval between the last context propagation and the crash of the primary server[2]. For these uncertain responses, there is a clear choice for the new primary that takes over a client: it can either transmit the response (risking the client seeing a duplicate if in fact the response had been sent before by the previous primary), or it can not transmit (risking that the client never sees the response). The choice is application specific. For example, for MPEG-encoded video, one would favor duplicate delivery for full image (I) frames over the risk of losing them, but would risk missing some incremental (P or B) frames.

Combining the observations above, we see that there is an interplay between the configurable factors of the frequency of propagation of the unit database information among the content group, and the number of members in each session group. The probability of losing context updates sent by the client is the chance of every session group member failing or separating from the client during the period between propagations. Thus this probability decreases as either the propagation frequency or the size of the session group rise. However, increasing either of these factors places more work on each server. Whenever client database information is propagated, each server in the content group must process it; when the session groups become larger, each server is a backup in more groups, and must therefore receive more client requests (however, the work is merely receiving and recording the request; only the primary responds).

5 Conclusions

We have presented a framework for building highly available services which are characterized by unchanging server contents, and changing context relating to each separate session. The framework is based on replication of the content among a group of servers. The context information is also replicated, but in three different levels of synchronization: The primary server has accurate information. The backups have somewhat dated information about which responses the primary sent, but accurate knowledge of the context updates sent by the client. The rest of the replicas have somewhat dated knowledge of the context. GCS is used for messages from the client to the service, so that the client can ignore issues of changes to the set of servers, and hand-over when a server fails, or when load is redistributed. GCS is also used to propagate information about the context from the primary to other servers. The key configurable parameters in our framework are the number of servers at each level of synchronization, and the frequency with which the primary propagates context to the other servers.

The framework of this paper is a generalization of the design used in the VoD service of [2]. Our description maintains the essential character of the earlier VoD design, with process groups at three scales. This paper extends the [2] work by making the configurable aspects explicit, and by introducing backup servers within the session group (giving an intermediate level of context synchronization between the up-to-date primary and the content group which receives propagated context from the primary).

Furthermore, we have analyzed the framework, to show which patterns of faults can leave the service not available to a client. We have shown where different properties of GCS are needed in the design, to allow consistent decisions. We have examined the impact of the configurable parameters on the chance of losing availability, and we have explained the tradeoffs between availability and performance.

Future work may integrate into the design some dynamic changes of the parameters, and automatic invocation of new servers using the techniques of [5]. Thus the user might express a desired service quality in terms of a chance of losing a context update, and the system could then adjust the needed number of backups in each session group.

Another extension worth pursuing is to integrate into the design a mechanism for consistently updating the state that is shared between clients, using the well-known replicated state machine technique [6].

[2] Recall that, unlike context updates caused by messages from the client, information about responses sent is not known to the backup servers in the session group, since we use point-to-point communication from the primary to the client.

References

[1] ACM. *Commun. ACM 39(4), special issue on Group Communications Systems*, April 1996.

[2] T. Anker, D. Dolev, and I. Keidar. Fault tolerant video-on-demand services. In *19th International Conference on Distributed Computing Systems (ICDCS)*, pages 244–252, June 1999.

[3] K. Birman. *Building Secure and Reliable Network Applications*. Manning, 1996.

[4] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32:374–382, April 1985.

[5] S. Mishra and G. Pang. Design and implementation of an availability management service. In *19th International Conference on Distributed Computing Systems (ICDCS) Workshop on Middleware*, pages 128–133, June 1999.

[6] F. B. Schneider. Implementing fault tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[7] R. Vitenberg, I. Keidar, G. V. Chockler, and D. Dolev. Group Communication Specifications: A Comprehensive Study. Tech. Report MIT-LCS-TR-790, MIT Lab. for Computer Science.