Data Replication under Latency Constraints

Siu Kee Kate Ho (siho@cs.brown.edu)

Abstract

To maintain good quality of service, data providers have to satisfy requests within some specified amount of time. One cheap way to achieve this is to place replicas of data objects so that clients could obtain requested data from at least one replica within given time limits. The number of these replicas should be minimized to reduce storage and update costs. In this paper, naïve and smart algorithms to determine placement of replicas are being implemented and investigated. In naïve algorithm, when current latency constraint cannot be satisfied, a new replica is placed in the server closest to the request client. In smart algorithm, if an existing replica can be moved to a new location and satisfy the new request, and if this move would not dissatisfy any existing request, this replica will be moved; otherwise, a new replica will be created in the server closest to the request client.

1. Introduction

Internet and other large-scale data dissemination systems are becoming increasingly important in our daily lives, for it can spread information widely, conveniently, and almost immediately. Often, we want to deliver data within some specific time, and having copies of data objects in different locations is one way to ensure timely service to clients.

Replica placement to satisfy latency constraints would be useful in applications such as content delivery network, which main concern is to deliver data to clients quickly and efficiently. It would also be useful in any large-scale database or data storage system, if data has to be extracted in a timely manner. This project is part of SAND[1], a scalable network database system using overlay topology.

1 SAND: a Scalable Network Database, by Yanif Ahmad and Ugur Cetintemel

1.1 Latency Constraints

With the growing importance in efficiency, fast and stable performance in data delivery becomes one of the clients' major concerns. One way to ensure stable delivery performance is to set up latency constraints, so data will arrive within a specific time limit. Different queries have different requirements on these latency constraints, depending on their functions and their values to the clients. However, most of the time, clients will sent out similar queries over a period of time, hence, latency constraints are specific to client, for particular data object.

1.2 Need for Data Replication

To satisfy the latency constraints mentioned above, we could either vary the speed of data transfer, or shorten the transfer distance. Since bandwidth and CPU speed are difficult and expensive to change, shortening transfer distance by placing replicas of data objects closer to requesting clients is the cheapest and essential ways to ensure data delivery time.

There are, however, some problems that arise with the increase number of replicas [21]. One of such problems is the increase in update cost. When a replica is created, an extra message to this new replica will have to be sent every time data is being updated. So, update cost increase proportionally with the increase number of replicas.

Another problem is data consistency. In content delivery network, where update occurs only on a small set of primary replica, this would not be a big problem. However, in a system where update can be recorded in any replica, the higher the number of replicas, the harder it is to keep data consistent, simply because there are more locations to worry about.

One way to mitigate these problems is to reduce the number of replicas created.

1.3 Approach

The main issue being explored in this paper is how to determine the placement of replicas dynamically, when latency constraints cannot be satisfied. The replica should be placed so that future requests can be answered within some given time limits. We implement two algorithm: naïve and smart. The naïve algorithm places a new replica in the server closest to the client when latency constraint cannot be satisfied. The smart algorithm considers moving an existing replica to satisfy this new latency constraint without invalidating old ones. By comparing these two algorithms, the tradeoff between calculation time and optimality is discussed. To ensure scalability, we based our design on an overlay location and routing `infrastructure` called *Tapestry*[3], which maintains decentralized resources.

1.4 Outline

The rest of the paper is organized as follows: in the next section we compare our algorithm against other related work. Tapestry is presented in more detail in section 3. In section 4, we describe our naïve and smart algorithm. In section 5, we discussed about clustering in Tapestry, and using it to improve our algorithm. In section 6, we evaluate these two algorithms. We will then outline future work in section 7. Finally, we conclude in section 8.

2. Related Work

Data replication is a popular topic, and a lot of researches had been done in this area. However, most of these focused on load-balancing or data consistency, and not many on latency constraints.

Many web content delivery uses DNS-based redirection to route clients' requests to the closest replica, so latency may be reduced. However, because of its centralized nature, its can easily result in server overloading, thus hindering the scalability [17].

The most similar work to this is probably Dynamic Replica Placement for Scalable Content Delivery [8]. The above paper proposed building dissemination tree built on top of Tapestry infrastructure to determine replica placement. It aims to balance load as well as satisfying latency constraints, with more focus on balancing load. It also proposed naïve and smart algorithm – the naïve one will place replica on the server closest to the client that satisfy all both load and latency constraints; the smart one looks into a larger set of server, and choose the one with lightest load, and if none satisfy, a replica is placed as far away from the request client as possible. Since its main concern is load balancing, its smart algorithm would create a lot of unnecessary replicas when resource is spare and load is balanced. Also, since it only create replicas, but never migrate or delete them, it will results in more replicas than our smart algorithm in the long run.

3. Tapestry Infrastructure

Tapestry is a distributed, scalable, fault-tolerant and self-organizing overlay location and routing infrastructure [3,4]. It locates data object quickly with guaranteed success. Unlike traditional models, which keep object locations in a centralized manner, Tapestry stores locations of replicas for different data objects in different servers. We refer these servers as nodes.

Nodes and objects in Tapestry can be identified with unique identifier, which is represented by a string of digits. Node identifiers are referred as node-IDs and object identifiers are referred as global unique identifiers (GUIDs). Every object is mapped to a node, which node-ID has the most common suffixs as the GUID of the object. We refer this node as the root node. The root node stores all the locations of the related object replicas.

Figure 1 shows a portion of Tapestry.

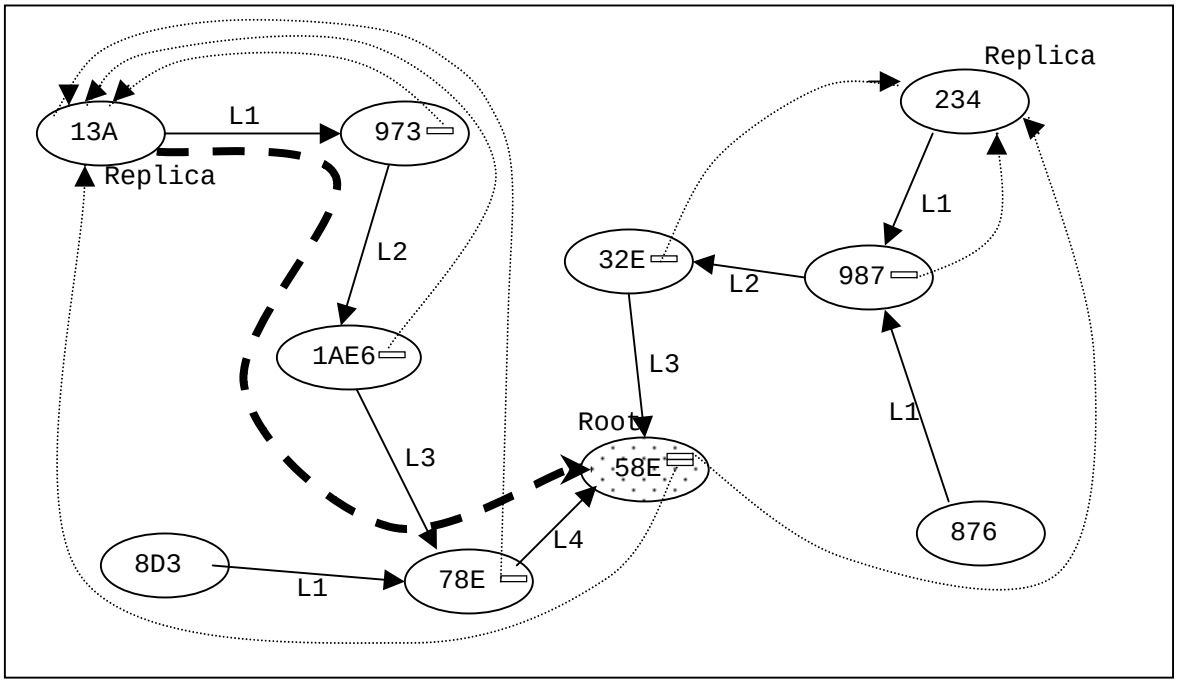


Fig. 1. The Tapestry Infrastructure

3.1 Routing

Tapestry uses local neighbor tables to route towards the root node. The neighbor table of a node is formed when this node is inserted. The table contains the node-IDs of some n nodes within some latency bound. The routing process tries to resolve destination address (which is GUID) one digit at a time, from right to left, each time closer to the root node. In figure 1, solid thin lines are neighboring links that connects one Tapestry node to another. As shown, digit is resolved one digit at a time, going from **** → ***6 → **E6 → *8E6 → 58E6.

3.2 Locating Replica

When a new object o is inserted or a new replica of o is created, the server s storing the object will send a message towards the root node for that object, leaving a location pointer in the form of <GUID(o), node-ID(s)> at each intermediate node. These are not a copy of the data object, but pointers to server s where o is being stored. When a node stores location pointers for multiple replicas, it keeps these pointers sorted in the order of distance from itself.

To locate a replica, the request client c sends a message towards the object root node. When it encounters a location pointer, it follows this pointer and route directly to the object. If no location pointer is encountered in any immediate node, it will eventually reach the root, where pointers to locations of replicas will definitely be encountered. Otherwise, this object does not exist.

Experiments in [20] shown that Tapestry can route any request from node c for object o, to the statistically closest node that contains a replica of o. It is shown that any existing node in the system will be found within at most logN hops, where N represents the size of ID namespace. In [12], it is shown that average distance traveled in locating an object is proportional to the distance from that object in terms of the number of hops traversed.

4. Replica Placement

We present two algorithms that dynamically place replicas with only limited knowledge about the network topology. Our goal is to satisfy latency constraints while minimizing the number of replica created.

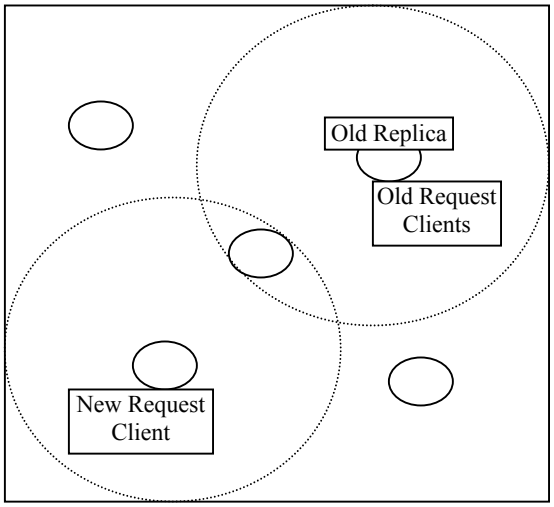


Fig. 2.1: Before applying any algorithm. In this diagram, dotted lines represent latency constraints, and solid line circles represent Tapestry servers.

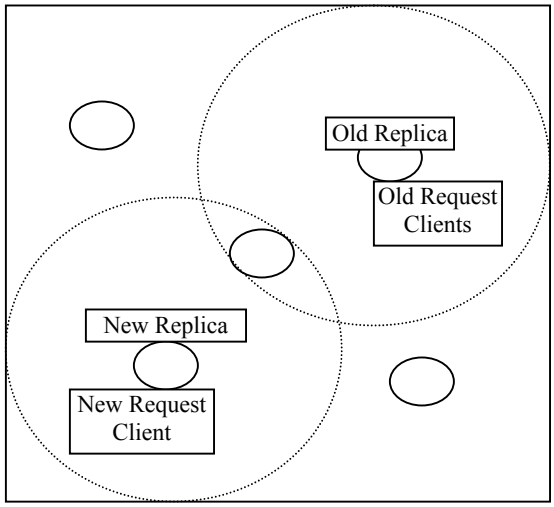


Fig. 2.2: After applying naïve algorithm

4.1 Naïve Algorithm

In this algorithm, we attempt to place a replica in the server as close to the request client as possible. We assume that any client c maintains a set of servers closest to itself, in order to connect to the rest of the Tapestry system. Hence, we simply place a replica of the requested object on the server the client contacted, when latency constraint cannot be satisfied.

Figure 2.2 illustrates this naïve algorithm.

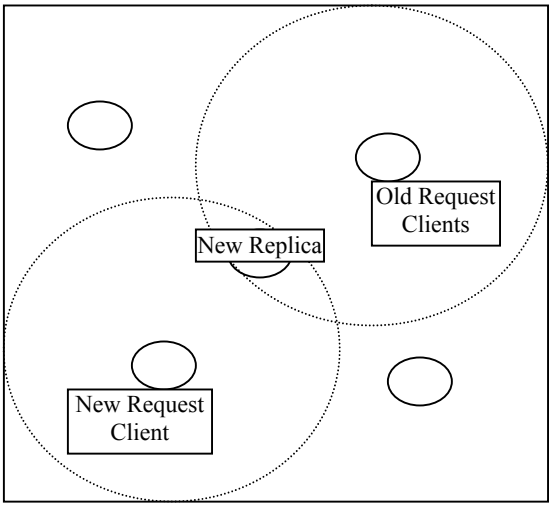


Fig. 2.3: After applying smart algorithm

4.2 Smart Algorithm

In smart algorithm, we attempt to move an existing replica to a new location so that it can satisfy the new request as well as all the existing requests. If there are multiple possible placement locations, the closest one from the request client will be chosen. This is illustrated in figure 2.3. If none of the existing replicas could be moved, we will then create a new replica in the server closest to the client, similar to the naïve placement. Comparing figure 2.2 and 2.3, it is obvious that smart algorithm will result in less number of replicas comparing to the naïve algorithm.

There is a risk with this algorithm if not implemented correctly: since replicas are migrated instead of created, if old supported clients are not checked carefully, the replica may be moved every time a request is made. For example, if client A makes a new request for object o, which is currently located in node O1, the smart algorithm relocate object o to O2. Client B then make a request for object o, and if the placement process does not check whether client A can still access object o within its latency constraint, the replica may be moved back to O1, and this would become a cycle. Hence, we have to be very certain that no currently supported clients will be dissatisfied after the replica is moved.

In order to save "ping" message, we try to use overlay latency to estimate the IP latency whenever possible.

```
Procedure NaiveReplicaPlacement (requesting_client c, object o)
1. Locate the closest replica r using Tapestry routing service.
2. if time(above_operation) ≤ latency_constraint then return r; exit
3. if pings(c, r) ≤ latency_constraint then return r; exit
4. s = first Tapestry server c connected to
5. puts a replica on s
6. s publishes o in Tapestry
7. return s
```

Algorithm 1: Naïve Replica Placement

```
Procedure SmartReplicaPlacement (requesting_client c, object o)
1. Locate the closest replica r using Tapestry routing service.
2. if time(above_operation) ≤ latency_constraint then return r; exit
3. if pings(c, r) ≤ latency_constraint then return r; exit
4. R = Ø
5. c sends a messages through Tapestry to root node of o, getting first 5 location
   pointers, insert these pointers to R
6. A = Ø
7. foreach re ∈ R
     a. if pings(c, re) ≤ 2*average_latency then
     b.     P = Ø
     c.     c sends a message through Tapestry to re, foreach intermediate node i
     d.         if (dist_Overlay ≤ latency_constraing) then insert i to P
     e.     N = P
     f.     foreach supported_client su in re
     g.         re sends message along with P to su
     h.         other = false; C = Ø
     i.         if su can be satisfied by other replica except re then other = true
                else
     j.             for each p ∈ P
     k.                 if pings(su, p) ≤ latency_constraint(su) insert p to C
     l.         N = N – C
     m.  A = A ∪ <re, C>
8. if A = Ø then put replica on server s closest to c; publish s, return s; exit;
9. old = null; new = null; sd = ∞
10. foreach a ∈ A
11.     if pings(c, getfirst(C)) < sd then
12.         old = re; new = getfirst(C)
13. move replica from old to new; publish new; unpublish old; return new
```

Algorithm 2: Smart Replica Placement

In the smart algorithm above, we assume that each node storing a replica maintains a list of last n clients accessing it for data in some extended period of time.

5. Clustering in Tapestry

As seen from the smart algorithm above, a lot of messages are being sent to check whether latency constraints of existing queries would be dissatisfied when a replica is being moved. This is highly inefficient since almost every node in a close-by area receives the same messages and does the same calculation. It would be more efficient if a node could represent all nodes in nearby area and only do the calculation once. One way to achieve that is by clustering.

5.1 Division of clusters

The locality property of tapestry ensures that nearby nodes often have common suffix. Hence, we can use this property to divide clusters without extra calculation. All nodes with the same l suffixs will be put into one cluster. In each cluster, a cluster head is chosen by routing towards destination *****sss, where s represents the suffix, and * are filled with the digit representing the number of suffix matched. For example, for cluster with last three suffix being 524, its cluster head is 33333524, filling all space except the suffix with 3. If the number of common suffix exceeds the base of the mesh digit, then (number of common suffix) mod base is being used as filling digit.

Moreover, this clustering system contains hierarchy of clustering, simply by specifying l. Cluster in level 0, in which no suffix has to be in common, contains the whole network. Clusters in level 1, in which 1 digit of suffix has to be in common, contains a large number of nodes. Clusters in level n, where n is the number of digits in node-ID, contains only the head node itself. The higher the level number, the less is the number of nodes contained within a cluster. In this division, we can ensure that every cluster contains several completed sub-clusters.

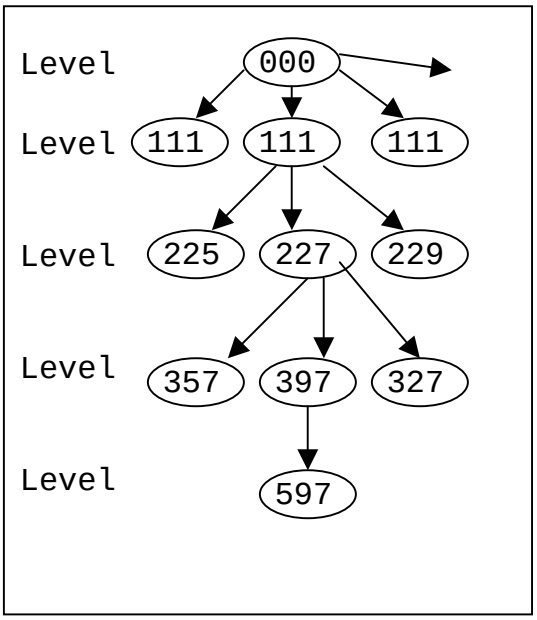


Fig. 3: Hierarchy of clients. Cluster heads are used to represent clusters in the above diagram.

Each replica head stores mindistance = minimum (latency_constraint(c) – distance (latency(c), headnode)), where c is any node within the cluster.

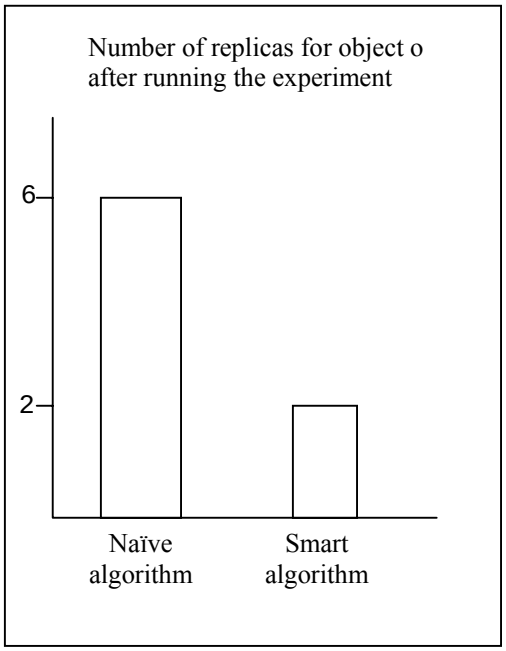
To apply clustering into the smart algorithm, we can change step 7f to 7k in algorithm 2 so that instead of going through every single clients, it checks, from cluster level n/2, n being the maximum number of digits in node-ID, whether mindistance - dist(head node, location-to-check-with) > 0. If it is, then all nodes within that cluster is satisfied, otherwise, it has to go down a hierarchy, and re-do the above.

6. Evaluation

Looking at the naïve and smart algorithm, we predict that smart algorithm will create less replicas compare to naïve algorithm, which seems to be the case looking at experimental result. However, the calculation cost for smart algorithm is also much higher than naïve algorithm.

We conduct our experiment with 30 Tapestry nodes. An object is first published from a random node. Afterwards, within some specific time limit, every node in the system request for that data object within some

latency constraints in a random time. Smart algorithm is being run several times to ensure that all clients' requests have really been covered. The results are presented in fig. 4.



Although smart algorithm produces much better result, its calculation cost is high, and our experiment shows that there is a decent chance that a replica may have to be moved multiple times before it reaches its optimal location. So, if latency constraints and network structure changes very often, there is a high chance that the replica will continuously be moving around and process time will be unnecessary wasted on calculating placement location and migrating replicas. In which case, using a naïve algorithm could be a better idea, when placement calculation and replica migrating costs outweigh the update costs. Moreover, having more replicas in different locations also means that the chances of satisfying latency constraints of new queries increases. However, when network structure is relatively stable and if latency constraints are not changed very often, then, it is best to use the smart algorithm, so update costs could be reduced. To determine which algorithm to use, one has to calculate update and placement costs for each algorithm, and choose the algorithm with least total cost.

7. Future Work

In this paper, only latency constraints are being considered. In the future, we would like to include load-balancing as one of the consideration as well. Moreover, more experiments can be conducted to compare the efficiency of different data replication algorithms purposed by others in different environments. Furthermore, algorithms for replica deletion can be explored to supplement with current replication placement algorithm, for better update performance. Last but not least, we can explore auto-selection of different replica placement algorithms based on local environment.

8. Conclusion

Techniques for placing and migrating replicas are being explored in this paper. We found that the smart algorithm which result in optimal replica placement maybe too costly in a fast-changing environment, when the costs in calculating and migrating replica outweigh data update cost. Hence, different algorithms should be applied in different environment.

Reference

1. Yanif Ahmad and Ugur Cetintemel. SAND: a Scalable Network Database.
2. Kubiatowicz, D. Bindel, et al. OceanStore: An Architecture for Global-Scale Persistent Storage. Proceedings of ACM ASPLOS, November, 2000
3. Ben Y. Zhao, et al. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. UCB Tech. Report UCB/CSD-01-1141
4. Kirsten Hildrum, et al. Distributed Object Location in a Dynamic Network. Proceedings of the Fourteenth ACM Symposium on Parallel Algorithms and Architectures (SPAA) 2002
5. Ben Y. Zhao. A Decentralized Location and Routing Infrastructure for Fault-tolerant Wide-area Applications
6. Sean Rhea et al. Pond: the OceanStore Prototype. Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST '03), March 2003
7. Dennis Geels and John Kubiatowicz. Appears in Proceedings of the SIGOPS European Workshop 2002. Replica Management Should Be A Game. Proceedings of the SIGOPS European Workshop 2002, Sep 2002
8. Yan Chen, Randy H. Katz and John D. Kubiatowicz. Dynamic Replica Placement for Scalable Content Delivery. Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS 2002), March 2002.
9. Patrick R. Eaton. Caching the Web with OceanStore. U.C. Berkeley Master's Report, Technical Report UCB/CSD-02-1212: November 2002.
10. Dennis Geels. Data Replication in OceanStore. U.C. Berkeley Master's Report, Technical Report UCB//CSD-02-1217, November 2002.
11. A. Rowstron et al. Storage Management and Caching in PAST, A Large-Scale, Persistent Peer-to-Peer Storage Utility. In Proc. Of the 18th ACM Symposium on Operating System Principles (SOSP), 2001.
12. G. Plaxton et. al. ALMI: Accessing nearby copies of replicated objects in a distributed environment. In Proc. Of the SCP SPAA, 1997.
13. I. Stoica et al. Chord: A scalable peer-to-peer lookup service for Internet applications. Proceedings of ACM SIGCOMM, 2001.
14. Yan Chen et al. SCAN: A Dynamic, Scalable, and Efficient Content Distribution Network. In Proc of the International Conference on Pervasive Computing, August 2002.
15. J. Jannotti et. al. Overcast: Reliable multicasting with an overlay network. Proceedings of OSDI, 2000
16. S. Jamin et al. Constrained mirror placement on the Internet. Proceedings of IEEE Infocom, 2001
17. T. Loukopoulos et al. An Overview of Data Replication on the Internet.
18. J. Wang. A Survey of Web Caching Schemes for the Internet. ACM Computer Communication Review (CCR), 29(5), 1999.
19. Guillaume Pierre. A Peer-to-Peer System to Bring Content Distribution Networks to the Masses.
20. Y. Chen et al. Quantifying network denial of service: A location service case study. Proceeding of Third International Conference on Information and Communications Security, 2001
21. J. Gray et al. The dangers of replication and a solution. In Proc. Of ACM SIGMOD Conf., 1996