Efficient Synchronization of Replicated Data in Distributed Systems

Thorsten Schütt, Florian Schintke, and Alexander Reinefeld

Zuse Institute Berlin (ZIB)

Abstract. We present *nsync*, a tool for synchronizing large replicated data sets in distributed systems. *nsync* computes nearly optimal synchronization plans based on a hierarchy of gossip algorithms that take the network topology into account. Our primary design goals were maximum performance and maximum scalability. We achieved these goals by exploiting parallelism in the planning and the synchronization phase, by omitting transfer of unnecessary metadata, by synchronizing at a block level rather than a file level, and by using sophisticated compression methods. With its relaxed consistency semantic, *nsync* neither needs a master copy nor a quorum for updating distributed replicas. Each replica is kept as an autonomous entity and can be modified with the usual tools.

1 Introduction

Global scientific collaborations in the fields of particle physics, earth observation, and life sciences, besides others, produce large amounts of data that are typically stored in distributed repositories and are accessed by thousands of researchers from many different locations [4,12,13,16]. Such environments require efficient, robust and scalable tools for managing the distributed data. Replication [14], which is often used to improve data availability, reliability and access latency, makes the data management task even more complicated.

We present a tool named *nsync* that helps in keeping replicated data in a consistent state. *nsync* was designed as a building block for a comprehensive data grid system such as GridLab [11], Globus [10] or SRB [2]. Our primary design goals have been efficiency and scalability. Following this goal, *nsync* uses hierarchical gossip schemes for computing nearly optimal synchronization plans. Moreover, *nsync* is itself implemented as an asynchronous distributed algorithm with full duplex communication.

In large distributed environments with many different administrative domains no assumptions can be made on the local software environments or file access methods. *nsync* allows all participating nodes to be completely autonomous in their local file management. No central server is needed and no replica location service is required. Files are maintained in their native file systems and therefore can be accessed and modified with the standard tools and programming interfaces.

nsync can also be used in large PC farms for software installation management. Especially in heterogeneous farms with hundreds or thousands of nodes, it is currently required to set up separate installation images for each different hardware and software environment. Using *nsync*, no images are needed. Any node in the cluster can be modified, and all changes will be propagated to the other nodes on demand.

P.M.A. Sloot et al. (Eds.): ICCS 2003, LNCS 2657, pp. 274–283, 2003.

The *nsync* software package [15] consists of 15,000 lines of C++ code using pthreads, network sockets (TCP), and the xerces XML library. It has the following features:

Highly scalable: *nsync* is implemented scalable, exploiting parallelism wherever possible. The data synchronization and its preparation, for example, are both performed by a distributed gossip algorithm.

Open and portable: All operations are performed on native file systems. Between two synchronizations, the repositories (i.e. the normal files in the systems) are kept independent. All updates are propagated in a single synchronization run. There is no master copy. In case of differing changes to the same file, the user is asked to resolve the conflict.

Topology-aware: To minimize synchronization time, *nsync* takes the physical network layout into account. Links with a higher bandwidth are preferred to those with a lower bandwidth. Network characteristics can either be specified in a configuration file or by linking monitoring tools like the Network Weather Service (NWS).

Transactional: *nsync* synchronizes on user request (offline synchronization). Conflicts are reported by *nsync* and must be resolved by the user.

Efficient: *nsync* transmits only modified blocks rather than complete files, and it uses efficient data compression algorithms to reduce the communication time.

User privileges: *nsync* is modest with respect to the required execution privileges. It can be installed and run with normal user privileges; only ssh login is required.

In the following, we present the design goals and the underlying algorithms of *nsync*. We discuss how *nsync* takes the network topology into account and demonstrate its scalability with some benchmark results on a large PC cluster. We conclude the paper with a review of related work and a brief summary.

2 Terminology

nsync synchronizes distributed *repositories*, that is, collections of replicas stored in geographically dispersed nodes. A repository may be a subtree of a local file system or a selection thereof, specified by a regular expression.

sync-world denotes all repositories that shall be kept in a consistent state. When invoking *nsync*, it synchronizes all repositories of a sync-world, leaving the files in a consistent state after the successful run. Note that a single file may be part of several sync-worlds at the same time. Conversely, multiple repositories of a sync-world may reside in a single node.

nsync handles files, directories and symbolic links, but skips sockets and pipes. In the following we use the term *file*, denoting all mentioned types. All repositories of a sync-world are specified in a configuration file.

The synchronization process is started by invoking the *sync-manager* which starts instances of *nsync* on all nodes in the sync-world. The synchronization is done in *rounds* comprising several node-to-node synchronizations that are performed concurrently between disjoint nodes. Each node is engaged in one node-to-node synchronization at a time, but different synchronization rounds may overlap in an asynchronous manner.

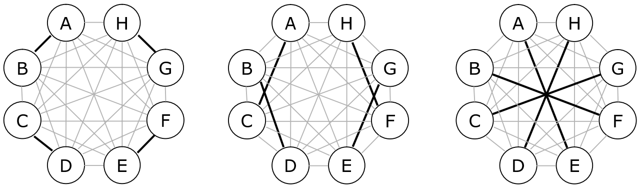


Fig. 1. Three rounds of n2n syncs can synchronize eight nodes.

3 The *nsync* Algorithm

The synchronization is done in two steps, the first for detecting files that have been modified since the last run, and the second for propagating the updated data. In the worst case, i.e. when all nodes have updated files, an all-to-all communication is needed.

In a naive approach, each node would send its updates to all partners, resulting in $(n-1)n/2$ open network connections in the system. Each single connection would trigger an independent local disk access, provoking many disk head movements and therefore resulting in a slow data transfer rate. To avoid this, *nsync* uses only node-to-node (n2n) syncs. Each node participates in at most one n2n sync at a time. Therefore, at most $n/2$ n2n syncs are run concurrently. Note that in each n2n sync a node propagates not only the modifications made to its own data, but also the modifications it received from other nodes in earlier rounds of the same synchronization. In total, each node needs a maximum of $\log_2(n)$ n2n syncs in a complete graph; see the discussion below.

The synchronization process is given by a list of rounds of parallel n2n syncs. No barrier operation is executed between the rounds and therefore rounds may overlap. Figure 1 depicts the synchronization of eight nodes. The process is split into three rounds. Each of them contains four parallel n2n syncs: (A↔B, C↔D, E↔F, G↔H), (A↔C, B↔D, E↔G, F↔H), (A↔E, B↔F, C↔G, D↔H).

3.1 Phases

The synchronization process is split into seven phases:

1. **Configuration:** The configuration file is read and data structures are initialized.
2. **Startup:** Instances of *nsync* are started on all participating nodes via ssh.
3. **Checking for Changes:** Each node determines all changes since the last sync and sends an estimation on the expected data transfer size to the sync-manager.
4. **Planning:** The sync-manager calculates a synchronization plan (reflecting the topology) and broadcasts it to all nodes.
5. **Dry Run:** The nodes simulate the synchronization plan without sending file contents to check for possible conflicts. Conflicts are reported to the user.
6. **Synchronization:** The synchronization plan is executed. Each node is engaged in one n2n sync at a time. Sync rounds may overlap.

7. **Finalization:** The sync-manager waits for all nodes to finish. All nodes update their local metadata in a two phase commit protocol.

In the following we explain each phase in more detail.

Configuration. The configuration of the synchronization environment is specified in XML format. For each sync-world the configuration file includes a description of the repositories and their hosts as well as a description of the network topology between the nodes. Each sync-world must have a unique `name`. An optional `pattern` can be used to select a subset of the files for synchronization. Each repository and its host is described by a `node` tag. The name of the `host`, the `path` to the repository, and the `username` for ssh are mandatory. The network topology is represented by a hierarchy of tags describing basic topologies like switched networks, hubs, rings, chains, meshes, tori and hierarchical compositions thereof.

Startup. The synchronization process is started by invoking the sync-manager on an arbitrary host. The sync-manager reads the configuration file and starts remote instances via ssh. Each instance allocates an Internet network socket and informs the sync-manager about its port number. All further communication between the instances and the sync-manager is done via this socket and is not tunneled via ssh for performance reasons. A future release of *nsync* will support ssl.

Checking for Changes. Each node maintains local *metadata* containing the name, type (file, directory or symbolic link), size, access permissions, and time of the last modification for the local files. At runtime all *nsync* instances check asynchronously whether changes have occurred since the last synchronization. This is done according to the following scheme:

Observed State	*Action*
New file on disk but not in metadata	Propagate file
File in metadata, but metadata differs	Propagate patch
File in metadata but not on disk	Propagate delete order

At the end of the checking phase each node has a list of changed files and also an estimation on the maximum data volume that needs to be transferred in the synchronization. Only the latter is sent to the sync-manager.

We do not need to synchronize clocks in a sync-world because the modifications are determined locally. When transferring a file from one site to another, the original time stamp is preserved. If the time zones of the source and target node differ, the time stamp will be adjusted accordingly.

Planning. The sync-manager calculates a synchronization plan and broadcasts it to the remote instances. In this calculation, it takes the communication bandwidths of the network links into account and computes an optimized communication sequence with a distributed gossip scheme (see Sec. 4).

Note that the sync manager has only an upper bound on the data volume that must be transferred, but it does not know about file names etc. Therefore, the planning phase is followed by a dry run, where the participating nodes check the synchronization plan for conflicts.

Dry Run. In the dry run, all nodes execute the synchronization plan, but without sending real data. This is necessary to detect conflicts between replicas that have been changed differently on different nodes. Such conflicts are reported to the user and must be solved manually.

Before reporting potential conflicts, false conflicts are sorted out. This is done by checking the conflicting replicas with MD5 checksums. If they were changed in the same way, i.e. have the same MD5 checksum, *nsync* will not report them as conflicting.

Synchronization. In the synchronization phase, the synchronization plan is executed on all nodes in parallel. Each node communicates with at most one other node in a round, thereby updating all files at both sides.

nsync uses different block transfer methods, depending on the available bandwidth between the nodes. For links with a low bandwidth *rsync* [17] and/or *bzip2* are used to reduce the data volume. On faster links, when the compression process would take longer than the data transmission, rsync and bzip2 are switched off.

Finalization. In the final phase, all instances report the successful completion of their synchronization steps to the sync-manager. The sync-manager allows the remote instances to update their local metadata and shuts down thereafter.

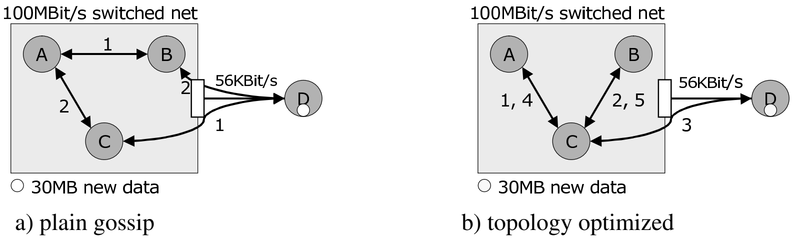
4 Topology-Aware Synchronization

For optimal synchronization based on n2n synchronizations gossip algorithms [1,5,7] can be used. The *constant model* takes only the startup cost of a connection into account. In the *linear model* the communication cost c is proportional to the size l of the data volume: $c = \beta + l\,\tau$, where β is the startup cost and τ is the transfer time of a unit-length message. τ is assumed to be constant for all links. Since *nsync* is designed for synchronizing large amounts of data, we use the linear model.

Determining a cost-optimal gossip plan for arbitrary graphs is an NP-hard problem [9]. We therefore simplify our algorithm by not supporting arbitrary graphs but only hierarchies of regular graph classes. Additionally, we treat network hubs like switches, thereby neglecting possible hub congestion. With these two restrictions, our plan generation algorithm has a runtime complexity of $O(n^2)$, but in practice often runs in $O(n \log n)$. It generates optimal plans for some classes of graphs [5,7], which are hierarchically composed by heuristics to allow for arbitrary networks.

4.1 Gossip in Complete Graphs

Complete homogeneous graphs can be used to model switched networks. For this purpose, we use the optimal algorithm described in [7]. It needs $\lceil \log_2 n \rceil$ rounds (resp. $\lceil \log_2 n \rceil + 1$ rounds) in graphs with an even (resp. odd) number of nodes. Depending on the amount of updated repositories, the synchronization time varies between $O(\log_2 n)$ for a one-to-all broadcast and $O(n)$ for an all-to-all broadcast. Similar algorithms for other graphs like rings and busses are also known [7], but they are only optimal in the constant model where the link bandwidth is ignored.



a) plain gossip

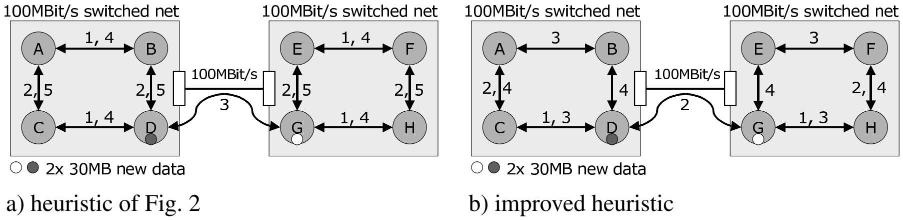
b) topology optimized

Fig. 2. Gossip considering bandwidth. Plain gossip needs $1280\,s + 1280\,s =$ **2560 s**. Optimized plan needs $2 \times 0\,s + 1280\,s + 2 \times 0.8\,s =$ **1281.6 s**.

4.2 Gossip in Practical Scenarios

Figure 2 illustrates the importance of considering the network characteristics in the synchronization process. Fig. 2a shows a straightforward but inferior gossip plan where the slow link between the left subnetwork and node D is used twice. *nsync* can do better. It treats this graph as a hierarchy of subgraphs. It regards nodes {A, B, C} as a combined 'meta node' and synchronizes it to D. More precisely, it first collects all updates of the subnetworks on their respective proxy nodes C and D (rounds 1 and 2), then synchronizes the proxy nodes via the slow link (round 3), and finally distributes the new data in each subnetwork (rounds 4 and 5).

Note that the described scheme is independent of the topology inside the subnetworks. The composition of gossips in the subnetworks can be done with algorithms that are specialized for rings, chains, grids, switches, etc.



a) heuristic of Fig. 2

b) improved heuristic

Fig. 3. Using two proxies (C & D; resp. G & H) reduces the WAN bottleneck: Plan (a) needs 5 × 2.4 s = **12.0 s**, plan (b) needs 4 × 2.4 s = **9.6 s**.

Figure 3 shows another example. In the left part, the scheme discussed above is applied to the two subnetworks of four nodes that are interconnected via a WAN link. In the right part (Fig. 3b), the local distribution of the data (round 1) is started immediately before the communication over the wide area link (round 2), so that the other nodes are able to distribute their data concurrently to the data transfer over the WAN, which is a bottleneck here. In effect, we have two proxy nodes in each subnetwork: nodes C and

D, and nodes G and H. One of each pair distributes the updates locally and the other manages the synchronization between the subnetworks. In detail, the synchronization in this example works as follows:

1. collect data on two proxy nodes (C, D and G, H) in each subnetwork (round 1)
2. synchronize between the proxy nodes over the WAN (round 2: D↔G) and start distribution in subnetworks (round 2: C↔A, H↔F)
3. continue to distribute data in each subnetwork (round 3: A↔B, E↔F) and distribute new data in subnetworks (round 3: C↔D, G↔H)
4. continue to distribute new data (round 4)

Note that much time is saved due to the overlapping rounds. Again this heuristic can be recursively applied for hierarchical topologies.

4.3 Combining Different Strategies

Sometimes it is difficult to determine which of the described strategies might perform better in practice. We therefore integrated a ‘simulator’ into *nsync* that estimates the runtime of alternative heuristics and selects the best.

For hierarchical topologies the above scheme is recursively applied [15]. The composition of multiple subplans is done with broadcasting and gathering operations. Our implementation uses gossiping for both. One nice feature of gossiping is the fact that it also generates optimal broadcast plans. If, for example, only one node has updated files, *nsync* eliminates the unused links in the planning phase, resulting in an optimal broadcast tree.

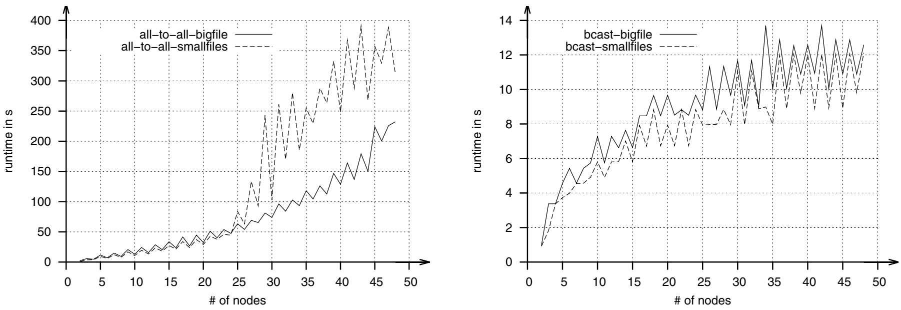
5 Empirical Results

We checked *nsync*’s performance on a PC cluster with 96 nodes, each equipped with two Intel Pentium-III processors of 850 MHz, 512 MB memory, and 4 GB disk space. The nodes are connected by a Fast Ethernet LAN with a CISCO 5509 switch with 3.6 Gbps internal bandwidth. Hence, the switch can be saturated with eighteen 100 Mb links in full duplex mode, and we therefore run our benchmarks only on up to 48 nodes (= 24 n2n communications). We evaluated four scenarios:

all-to-all-bigfile: A 10 MB file on each node must be propagated to all others.
all-to-all-smallfiles: Same as before, but the 10 MB is split in 1024 files.
bcast-bigfile: A 10 MB file on *one* node must be broadcasted to all others.
bcast-smallfiles: Same as before, but the 10 MB is split in 1024 files.

The results are illustrated in Fig. 4a for the all-to-all communications and in Fig. 4b for the broadcast scenarios. For each data point, we plotted the lower of two independent runs. We could do so, because the variance between the two runs is in the one-digit percent range with only some rare exceptions.

First, we notice the zigzag pattern in all graphs. It is caused by the gossip scheme, which needs one or two additional communication rounds in graphs with an odd number of nodes as compared to the graph with the next lower even node number.



a) all-to-all scenario

b) broadcast scenario

Fig. 4. *nsync* performance on a PC cluster.

In the *all-to-all-bigfile* scenario, the runtime of distributing the single 10 MB file to all other nodes grows constantly with an increasing number of nodes. At the very right side, with 45 nodes, we notice a slightly sharper rise. This is attributed to the limited main memory size in the nodes. With 45 nodes, a total of $45 \cdot 10\,\text{MB} = 450\,\text{MB}$ of data must be maintained in each node, which does not fit into the nodes' main memory of 512 MB (deduct some memory space for the system software).

When splitting the 10 MB file into 1024 files in the *all-to-all-smallfiles* scenario, the curve becomes less regular in systems with more than 25 nodes. Here the total amount of storage is not the limiting factor (it is the same as before), but rather the number of files. With 25 nodes in sync-world, each node must manage 25,600 files, including the corresponding file descriptors. Hence, the irregularly rising zigzag pattern at the right hand side of the all-to-all-smallfiles plot must be attributed to the Linux file handling and caching mechanisms.

In the *bcast-bigfile* and *bcast-smallfiles* scenarios shown in Fig. 4b fewer data is sent and so in general less runtime is needed as can be seen from the different scaling of the y-axis. Both curves rise with the logarithm of the number of nodes. This is as it should be, because the (virtual) broadcast tree that is built up, grows at a logarithmic scale.

Note that in the two broadcast examples, *nsync* performs better in graphs with an odd rather than an even number of nodes whereas in the all-to-all cases, it was the other way around. This is because in the broadcast scenario the result of the leaf nodes does not need to be propagated back to the root, thereby fewer rounds are needed.

6 Related Work

Several replica systems have been developed, mainly in the domain of distributed file systems where strict data consistency is of prime importance. AFS and NFS are such systems. They assume a much tighter coupling of the individual nodes than can be supported in geographically distributed Grid environments. Here, many different administrative domains (with different local policies) are linked to form a single environment. Moreover, in Grid environments the files are kept autonomously and they are less often updated from remote nodes. Hence a weaker consistency model suffices. The more flexible consistency models described in [8,18] do not match our requirements, because we aim at offline synchronization that allows user defined transactions.

In the field of loosely coupled synchronization systems for Grid environments *Rumor* [6] is most notably. It supports the synchronization between systems that are not always online. If a node is temporarily offline, it will be updated at a later time. For this purpose, Rumor provides a mechanism of versioning vectors.

For efficient broadcasts of the differences between files, *rsync*+ [3] can be used. It is based on a master-copy model. If one out of n copies has been changed, the differences are locally calculated (using only one other client and thereby reducing transfer traffic) and only the patch is broadcasted to all other nodes.

7 Conclusion

nsync is an efficient tool for file synchronization and distribution in Grid and cluster environments. It uses a hierarchy of gossip algorithms for computing a nearly optimal synchronization plan.

Our major goal in the design of *nsync* was to achieve maximum scalability. *nsync* can be used in environments with thousands of nodes and an order of magnitude more files. We avoided performance bottlenecks by exploiting node parallelism (even in the planning phase), by omitting the transfer of unnecessary metadata, by synchronizing on a block level rather than file level, and by using a variety of data compression methods. Note that in other synchronization software the amount of transmitted metadata (file names, data types, etc.) often becomes a bottleneck—especially in very large environments. *nsync* does not need this information, because the synchronization plan is calculated in two steps, a very fast sequential step for calculating the general synchronization pattern, and a second step for remotely detecting conflicts in parallel.

nsync has been designed as part of a larger research endeavor on the management of distributed data. It is a building block in the GridLab project [11], but can be similarly deployed in SRB [2] and Globus [10].

Acknowledgements. Special thanks to Axel Keller (Paderborn) for his help in the use of the PC^2 cluster. This research was partly funded by the EU GridLab project.

References

1. B. Baker and R. Shostak. Gossips and telephones. *Discrete Mathematics*, 2:191–193, 1972.
2. C. Baru, R. Moore, A. Rajasekar, and M. Wan. The SDSC Storage Resource Broker. In *Proceedings of CASCON'98*, Toronto, Canada, November 1998.
3. B. Dempsey and D. Weiss. On the performance and scalability of a data mirroring approach for I2-DSI. In *Network Storage Symposium*, 1999.
4. A. Chervenak et al. Giggle: A framework for constructing scalable replica location services. In *Proceedings of the SC 2002*, Baltimore, Maryland, November 2002.
5. P. Fraigniaud and E. Lazard. Methods and problems of communication in usual networks. In *Discrete Applied Mathematics*, volume 53, pages 79–133, 1994.

6. R. G. Guy, P. L. Reiher, D. Ratner, M. Gunter, W. Ma, and G. J. Popek. Rumor: Mobile data access through optimistic peer-to-peer replication. In *ER Workshops*, pages 254–265, 1998.
7. J. Hromkovic, C. Klasing, B. Monien, and R. Peine. Dissemination of information in interconnection networks. *Combinatorial Network Theory*, pages 125–212, 1995.
8. R. Jiménez-Peris, M. Patiño-Martínez, G. Alonso, and B. Kemme. How to select a replication protocol according to scalability, availability, and communication overhead. In *IEEE Int. Conf. on Reliable Distrib. Systems (SRDS'01)*, New Orleans, October 2001. IEEE CS Press.
9. D. W. Krumme, G. Cybenko, and K. N. Venkataraman. Gossiping in minimal time. *SIAM J. Comput.*, 21(1):111–139, 1992.
10. Globus Project. http://www.globus.org.
11. GridLab Project. http://www.gridlab.org.
12. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *Proceedings of ACM SIGCOMM 2001*, 2001.
13. M. Ripeanu and I. Foster. A decentralized, adaptive, replica location service. *In Proceedings of 11th IEEE Int. Symp. on High Performance Distributed Compuing (HPDC-11)*, July 2002.
14. F. Schintke and A. Reinefeld. On the cost of reliability in large data grids. Technical Report ZR-02-52, Zuse Institute Berlin (ZIB), December 2002.
15. T. Schütt. Synchronisation von verteilten Verzeichnisstrukturen. *Diploma Thesis*, 2002.
16. I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.
17. A. Tridgell. *Efficient Algorithms for Sorting and Synchronization.* PhD thesis, Australian National University, 1999.
18. H. Yu and A. Vahdat. The costs and limits of availability for replicated services. In *Proc. of the 18th ACM Symposium on Operating Systems Principles*, pages 29–42. ACM Press, 2001.