HADES – A Highly Available Distributed Main-Memory Reliable Storage

Matthias Meixner, Alejandro Buchmann
Datenbanken und Verteilte Systeme
Department of Computer Science, Technische Universität Darmstadt
Hochschulstraße 10, 64289 Darmstadt, Germany
E-mail: {meixner,buchmann}@informatik.tu-darmstadt.de

Abstract

Fast persistent storage is a requirement in many applications in which slow disk access times become the bottleneck. This paper describes HADES, a main memory storage system, that uses a fault tolerant partitioning scheme to reliably store data in main memory of a distributed computer system and is, therefore, able to improve performance. It presents the principles involved, discusses performance evaluation and compares HADES to other systems.

1 Introduction

We are starting to see new applications that make high demands on data storage systems regarding both availability and access time. Examples are highly available systems using IP take-over and publish/subscribe systems.

1.1 IP take-over

One practical way of achieving high availability in distributed systems is IP take-over (Fetzer et al., 2003). A service is bound to an IP address and port number. If the service fails, the client tries to reconnect, which is detected and another server takes over the IP address and continues to provide the service.

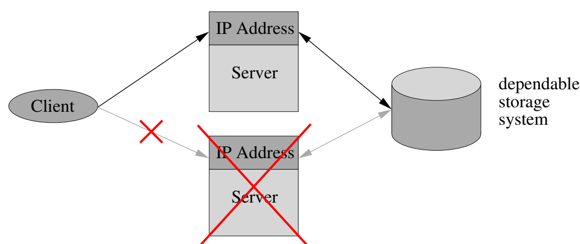

Figure 1: IP take-over

To process client requests correctly, most services need to keep some state. When a server crashes, state stored in memory is lost. One way of dealing with this problem is to source out state information to stable storage and treat the service as if it was stateless (figure 1).

The response time of this system heavily depends on the performance of the storage system. Each operation that changes the state that is associated with a client must go to the storage system before replying to the client so that another server could resume the operation in case of a failure. In this application it is of capital importance that the storage system is highly available, since the availability of the whole system depends on it.

1.2 Publish/Subscribe Systems

Publish/subscribe systems are used to build large distributed systems in which the components are loosely coupled (Eugster et al., 2003; Carzaniga et al., 2000):

- A component does not need to know which other components exist. It just needs to know about the format and structure of events that it is interested in.
- Components can be added and removed without directly affecting other components.

Components communicate via *notifications* that signal the occurrence of an event. Clients subscribe to notifications.Within the *event notification system event brokers* use this information to forward notifications from producers to consumers. Filters are used to avoid flooding and limit the forwarding to those consumers interested in certain notifications. Furthermore, filters may aggregate information (e.g. calculate min/max values over a period of time) or transform information (e.g. convert degrees Fahrenheit into degrees Celsius). Figure 2 gives an example of the flow of data from a producer to a consumer showing only the active components that are relevant for this flow of information. In this system there cannot be an end to end error correction, since a connection between the producer and the consumer does not exist and the producer does not know the identity or even the number of consumers. Therefore, within this system notifications may be lost or duplicated.

Figure 2: Flow of data in an event based system

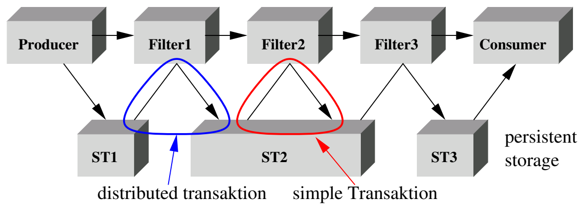

Figure 3: notifications and transactions

This property is not suitable for applications that depend on an “exactly once” delivery of notifications, e.g. applications that count the occurrence of certain events. The same applies to event aggregation that needs to store intermediate results without losing them. This problem can be solved by using transactions and persistent intermediate storage (figure 3). Notifications are passed on by transferring them from one persistent storage to another within a transaction. The atomicity of the transaction guarantees an exactly once delivery and the persistent storage guarantees that no notification is lost in case of a failure.

Using this approach poses some challenging requirements on the storage system: the end to end delay depends on the time needed for the data access required by the transaction, since the next filter in the path cannot start processing the data until the preceding operation has been committed. Therefore, the data access time is critical to the performance of the whole system. If the storage system goes down, the flow of notifications is interrupted until the storage system is restarted. To minimize this effect a storage system is needed that additionally provides high availability and reliability.

1.3 Common requirements

Both applications described above impose similar requirements on the storage system: the performance heavily depends on the access time which is dominated by latency. The amount of data to be stored is relatively small but data is modified frequently. For performance reasons we can assume that all data should be stored within the same LAN. Both applications require high availability of the storage system.

Conventional storage systems are not suitable to be used within these applications: the use of hard disks would be a bottleneck since even the fastest available disks have an access time of more than $5ms$. Flash ROM storage systems cannot be used due to the frequent modification of data: flash memory has a limited number of write cycles, e.g. 100000 for *Intel StrataFlash Memory (J3)*. DRAM based solid state disks do not have this limitation, but their cost is 2 orders of magnitude higher than that of conventional disks. For example for the price of an Athena-2 plus disk of 800MB one would easily get 8 complete computers with 8 GB of main memory in total. Furthermore, a single disk cannot provide high availability, therefore at least two of them are needed plus some RAID system that manages redundancy.

1.4 Aim of this paper

The aim of this paper is to describe how fault tolerant partitioning schemes can be used to provide fast reliable storage and improve the performance of the applications presented above. In section 2 we describe the implementation of HADES, a data management system, that offers fast access time and high availability through the use of fault tolerance. In section 3 we give a performance evaluation and in section 4 we discuss the features that distinguish HADES from existing systems. Finally we give a short summary and outlook in section 5.

2 HADES architecture

HADES follows the principle of achieving persistence by redundantly storing data in main memory of at least two nodes of a cluster. If a node fails, there is still a copy and data can be accessed without downtime. After a failure has been detected, redundancy is restored so that another failure can be masked.

2.1 Fault model

The fault model in HADES is based on is *Crash* (Gärtner, 1999), i.e. a computer operates flawlessly or not at all. While redundancy is reestablished after a failure, no other node in the cluster may fail. Node failures are detected using a distributed heartbeat based fault detector.

It is assumed that cluster nodes fail independently of each other. Therefore, all nodes should be guarded against power failure using an uninterruptable power source. A redundant network interconnection prevents that the connection to more than one computer is lost in case of network problems. Since all nodes are within the same LAN this does not pose a big problem. One simple solution is to use one switch per node and redundantly connect these switches. The spanning tree protocol used in the switches deactivates redundant links and reactivates them in case of a failure. A failure of a switch is equivalent to the failure of one single node, a network split does not occur.

2.2 Addressing

One of the main problems is to efficiently address data in the cluster. Network addresses cannot be part of an addressing scheme, since the location where data is stored may change in case of a node failure. The addressing scheme needs to support redundancy in a way that makes it possible to access the data even after a crash of one node. It must support the reestablishment of redundancy after a crash, i.e. it must be able to change the node where it expects to find data. The addressing scheme should provide an efficient data access, i.e. in normal operation no more than one request should be required to read or write data.

slice	primary server	secondary server
0	A	B
1	A	B
2	B	C
3	B	C
4	C	A
5	C	A

→ Reestablishing of redundancy

slice	primary server	secondary server
0	A	B
1	A	B
2	B	A
3	B	A
4	A	B
5	A	B

→ load rebalancing

slice	primary server	secondary server
0	B	A
1	A	B
2	B	A
3	B	A
4	A	B
5	A	B

Figure 4: Data distribution: second stage

HADES solves this problem using the following addressing scheme: data is stored in pages that are numbered in ascending order $(0, 1, 2, \ldots)$. Unlike disk blocks or MMU-pages, pages in HADES are not of fixed length and may have an arbitrary size. Therefore, we can assume that one page holds one record without degrading the resource utilization.

HADES uses a two stage mapping from pages to nodes. The first stage maps page addresses to *slice addresses*, which is equivalent to a horizontal fragmentation of data:

$$slice_number = page_number \bmod slice_count$$

The slice count is constant and does not change even when nodes join or leave (fail) the cluster. It is specified at start time and it should be at least as high as the number of servers in the cluster considering possible future extensions. A higher number of slices results in a more uniform load distribution and offers better extensibility but at the same time generates a higher communication overhead.

The second stage uses a mapping table to map slice numbers to cluster nodes. It assigns a primary and secondary server to each slice. The secondary server acts as a backup server that can seamlessly take over the tasks of the primary server in case of a failure. The mapping adapts to achieve a uniform load distribution: each node is assigned not more than $2 \times \lceil \frac{slice_count}{number_of_nodes} \rceil$ slices (The factor of 2 is due to assigning a slice to two servers: primary and secondary server). When recalculating this distribution, after a node has failed or a new node has been added, HADES regards former data distributions and minimizes changes, thereby reducing the amount of data that needs to be copied to other nodes. Figure 4 gives an example of this: after a failure of node C data stored in slices 0 and 1 does not need to be copied, only data in slices 2-5 is copied to restore redundancy (highlighted in grey). A final swap of primary and secondary servers is used to achieve further load balancing between their roles as primary and secondary server. This mapping table is calculated and distributed by an elected coordinator, therefore problems with consistency cannot arise.

This addressing scheme offers several advantages: the first mapping stage is fixed and, therefore, it does not need to be transmitted to other nodes. The *mapping table* of the second stage needs to be updated only in case of a change of nodes due to node failure or insertion, in particular it does not require an update when inserting or deleting data. The mapping table is small (it has only $slice_count$ entries) and, therefore, it can be easily distributed to all other servers and clients. Using this mapping table clients are able to determine the storage location of data. Therefore, in normal operation only one request is required to access data.

2.3 Data consistency

As soon as several copies of data exist, there is always the problem of consistency: which copy of data is valid and may be accessed? To be used as a replacement for other storage systems, HADES provides similar guarantees regarding consistency: only write accesses may change data. While this sounds simple, it is not as the following example illustrates:

Client A writes page 1 which resides on server X. This page is read by client B. If server X fails before synchronizing page 1 with the backup server Y a subsequent read request of client B will read the previous content of page 1. In this case client B sees a change of data back to the old content although there was no write access. This must not happen.

To avoid these types of problems HADES uses one primary server for each slice (which may be different for different slices) that is responsible for maintaining data consistency. Clients communicate only with the primary server. This server coordinates read and write accesses and synchronization of data with the secondary server: write accesses are acknowledged only after the data has been successfully copied to the secondary server, read requests only read already synchronized data and, if necessary, they are delayed until the synchronization of the accessed page has been completed. The delay time is up to no more than half the time required for a write access (about $0.3ms$), i.e. delaying the request does not lead to performance problems.

If the primary server fails, the secondary server takes over the role of the primary server and the vacant role of the secondary server is assigned to one of the remaining nodes of the cluster. Note that nodes may play several different roles at the same time, e.g. primary server for one set of slices and secondary server for another set.

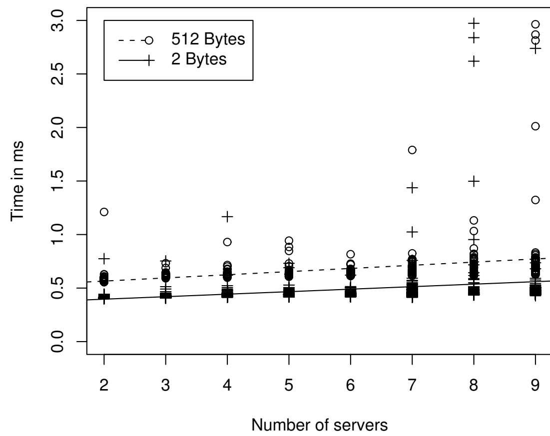

Figure 5: Writing data

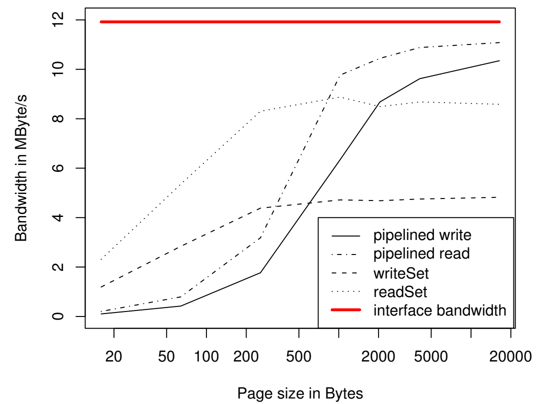

Figure 6: Bandwidth

2.4 Transactions

HADES has two modes of operation: with and without support for transactions. Using the mode without transactions, HADES can be used as a fast highly available replacement for hard disks. Using the mode with support for transactions HADES provides classical *ACID properties* (atomicity, consistency, isolation and durability (Gray, 1979)) to coordinate the simultaneous access of several clients or the support of reliable publish/subscribe. Transactions are controlled by an elected coordinator and backup coordinator. If a client crashes, transactions are automatically aborted.

HADES uses shadow paging (Lorie, 1977): changes are stored within separate memory regions and in case of a commit only a pointer needs to be switched. Since one page only contains one record, concurrency problems that result from storing several records in one block do not exist.

HADES supports distributed transactions spanning several clusters. This is realized by attaching *external* transactions to a normal transaction running on a different cluster thus forming one distributed transaction spanning several clusters. Control passes to the transaction to which an external transaction was attached. External transactions differ from normal transactions in that they cannot abort on their own in case of a failure of a client but only as part of the distributed transaction. This is required to prevent race conditions that only abort parts of the distributed transaction while others are committed if a client fails shortly after sending commit in the presumed commit two phase commit protocol used.

When an external transaction is attached to another transaction, the cluster containing the latter transaction becomes transaction coordinator for this distributed transaction and is responsible for either aborting or committing all parts of the transaction. The transaction coordinator is also responsible for initiating the abort of the distributed transaction in case of a failure of the client. Since HADES uses a complete cluster as transaction coordinator and the failure of one node does not affect the availability of the cluster as a whole, the use of two phase commit (Gray, 1979) is sufficient and three phase commit (Skeen, 1981) is not needed since no blocking can occur.

External transactions are created and attached to another transaction using only one atomic step. Therefore, there is no risk of client failure between the creation and attaching of the external transaction.

2.5 Optimizations

Operations in HADES are not limited to read/write. More complex operations and operations that operate in parallel on whole slices can be added to reduce network traffic and improve speed. One good example is searching data: instead of reading all the data and searching on client side, the search request is transmitted to the cluster and performed locally.

3 Performance results

A cluster of 10 identical computers with AMD Athlon XP2000+ and 1 GBytes of main memory running HADES on top of Linux at user level was used as a test environment. The computers were connected using fast Ethernet.

Several performance measurements of different operations and different configurations were done. Due to space limitations we selected some representative results.

Write operation
This operation was performed using page sizes of 2 and 512 bytes and using a different number of servers. The result using 512 bytes can be directly compared to hard disk accesses. For each number of servers 100 write accesses were performed and the time of each single access was measured. HADES reaches access times of about $0.6ms$ (figure 5) and is therefore, about one order of magnitude faster than the access time of the fastest hard disks that currently reach about $5.4ms$ (IBM/Hitachi Ultrastar: $3.4ms$ seek time, $2ms$ latency).

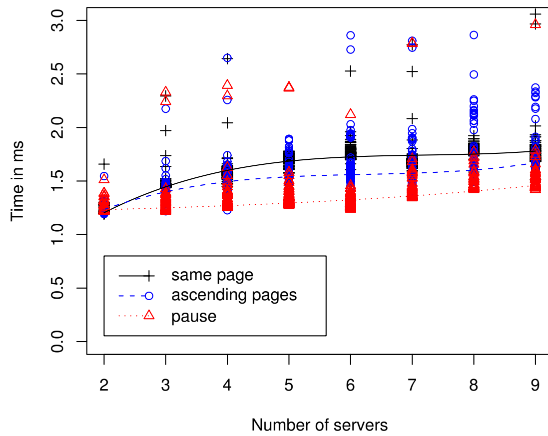

Figure 7: Transaction: read - modify - write

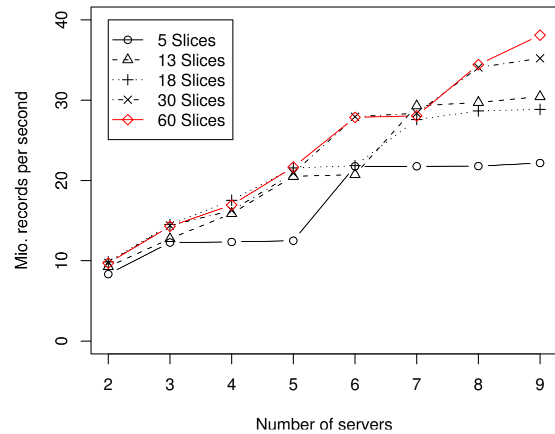

Figure 8: Tablescan performance

Bandwidth
Although not as important as the other operations for our target applications HADES supports optimizations of bandwidth: pages may be read and written in parallel (pipelined) or in sets of several pages. Figure 6 gives the results obtained using a single client using no transactions. Even for moderate page sizes of about 1024-2048 bytes the transfer speed is quite close to the maximum speed offered by the network interface.

Transactions
The scenario given in the introduction in which information is disseminated in reliable mode was evaluated. Within a transaction data is read, modified and written back. The transaction terminates with commit. Three cases were considered:

1. Always the same page is read and written. The next transaction, therefore, possibly needs to wait until all locks of the previous transaction are cleaned up and released.

2. A different page is read and written each time. Therefore, locks do not have any influence, but completing the cleanup of the preceding transaction may still consume some computing power.

3. A small pause is inserted between transactions, therefore, each transaction may use the full computing power.

The first two cases correspond to a system operating near full load, whereas the last one corresponds to a system that has some reserves left. HADES reached times around $1.5ms$ for the whole transaction (figure 7), i.e. the whole transaction can be completed faster than one third of the time required by one single access to hard disk.

Tablescan
Searching is a task that can be easily parallelized and therefore can utilize the parallel computing power of a cluster. This performance test used a table with 2.000.000 random entries (each of 48 bytes). A selection was performed that is equivalent to the following SQL-expression:

SELECT COUNT(*) FROM benchmark WHERE Category='a';

The measurement was performed using several slice counts to see the effect of the number of slices. Figure 8 gives the obtained results. Overall HADES scales quite well as long as there are enough slices to evenly distribute the load. Uneven load distribution can be seen as levels in figure 8. These levels result from the server with the highest load dominating the required time. The following example will illustrate this effect: when distributing 6 slices to 4 or 5 nodes, in both cases at least one node will be assigned 2 slices whereas the others will be only assigned one slice. The node assigned 2 slices will dominate the performance and therefore there will be no performance gain when going from 4 to 5 nodes. Altogether, the measured performance numbers indicate that the optimal number of slices should be around 3-4 times the expected number of servers.

4 Comparison with other systems

There are several systems that *seem* to be suitable as a storage architecture for the applications presented or look similar to HADES, but they all lack at least one important property.

4.1 Conventional Database Management Systems

Conventional Database Management Systems (DBMSs) store data on hard disks. Therefore, these systems are limited by the disk access. Although these systems try to reduce the effect of disk accesses by caching, they cannot totally suppress slow disk access times. Write accesses cannot be safely cached, data is stored safely only after it has actually been written to disk or the log has been flushed to disk. Notably short transactions consisting of few writes suffer most from disk accesses, since they cannot take advantage of the high bandwidth of modern disks but mainly depend on the access time, which is only very slowly improving compared to bandwidth and density.

To demonstrate this effect we did some performance evaluations using PostgreSQL and BerkeleyDB. PostgreSQL needed about $100ms$ to insert a single data item and BerkeleyDB about $51ms$ (using a standard IDE hard disk). So both were about two orders of magnitude slower than HADES. We also measured the search performance. PostgreSQL reached about 0.9 Mio. scanned records per second and BerkeleyDB – since BerkeleyDB only supports searching for keys but not for data we had to implement the searching within the application – reached about 0.4 Mio. scanned records per second compared to at least 8 Mio. records per second that were achieved by a HADES cluster of two nodes. This is due to the fact that PostgreSQL and BerkeleyDB are designed to take into account that data is stored on disk whereas HADES can directly access data in main memory.

Even main memory DBMSs are affected by hard disk access times: although main memory databases like e.g. PRISMA/DB (Apers et al., 1992) store data in main memory, they use hard disks for logging just like other databases do. Pre-committing and group commit (DeWitt et al., 1984) allow to improve throughput, but they cannot improve commit latency. Therefore, also in this case hard disk access limits the response time of transactions.

4.2 RAID

Using RAID systems is the standard way of improving disk bandwidth (Patterson et al., 1988). However RAID systems cannot improve latency. Since small reads and writes are dominated by latency RAID systems cannot improve performance in our case.

4.3 Distributed checkpointing

If a node fails in a distributed system all intermediate results are lost and the (probably expensive) computation needs to be restarted. To avoid a restart from the very beginning some systems regularly take checkpoints (snapshots) of the distributed system state and are therefore able to continue the computation starting from the last complete checkpoint. Similar to HADES, RDSM (Kermarrec, 1997) and (Plank and Li, 1994) keep all information in main memory. Different from HADES these systems can only guarantee persistence as soon as a new checkpoint has been taken and all modifications since then are lost. To reach the same degree of persistence of HADES using these systems a checkpoint would have to be taken as part of each write access, which would be prohibitively expensive.

4.4 Distributed hash table

Distributed hash tables (DHTs) try to redundantly store and locate information in a WAN using peer to peer techniques. Examples are Chord (Stoica et al., 2001), Pastry (Rowstron and Druschel, 2001), Tapestry (Zhao et al., 2001) and CAN (Ratnasamy et al., 2001). While both HADES and DHTs store data in a network of computers this is already where the similarities end, since the design goals were totally different. Chord, for example, was designed with scalability in a WAN in mind whereas HADES was tuned for access time in a LAN. HADES requires $O(1)$ messages to access data compared to $O(\log N)$ messages required by Chord. Chord uses a probabilistic approach for load balancing. While this works on a large scale it may lead to a very uneven distribution on a smaller scale. Therefore, HADES uses explicit load balancing, which can guarantee an even distribution of data. HADES supports both atomic insert/update and transactions none of which is supported by Chord. So although HADES and Chord might look similar at first sight, they have very different properties and fields of application. The same applies for other DHT systems.

Another example is the LH*m scheme (Litwin and Neimat, 1996). It extends linear hashing to a scalable distributed data structure: each bucket is stored on one computer per site. Two sites are used, one being the backup of the other. For load balancing also the clients are divided into two groups having computers from one site as primary servers and computers from the other site as secondary servers. Since LH*m does not support atomic insert/update and transactions it has problems regarding consistency: if the same data record gets written at the same time by clients having the primary server in different sites, the messages for copying the data record from one site to the other may cross each other and after this different versions of the same object exist in the different sites. As a result, reading the same record results in different data, depending on which site the client belongs to. This is not acceptable for applications we are considering.

4.5 Distributed write cache

LND (Mao et al., 2002) implements a distributed fault tolerant write cache to reliably cache data to be able to speed up operation by asynchronously writing data to disk. This approach does not support transactions and is limited to one client which represents a single point of failure. If the client fails a recovery phase is needed, therefore, unlike HADES, which can serve several clients in parallel, LND is not able to provide uninterrupted service.

5 Summary and outlook

HADES offers significant improvements regarding access time and is, therefore, in particular useful for reliable publish/subscribe systems, publish/subscribe systems that aggregate events and therefore must store state, and systems using IP take-over.

Further speed improvements can be expected by moving from Ethernet and TCP/IP to network technologies, that were designed to reach low latencies like SCI,

Myrinet or Infiniband.

Although no real-time operating system and network was used during the performance evaluation, the performance measures show an astonishingly low number of outliers (figure 5). Therefore, a tuning of HADES towards real-time databases seems promising.

But the mechanisms used by HADES are also very interesting for a very different field of applications: sensor networks (Akyildiz et al., 2002). Sensor networks follow the idea of having very small, battery powered sensor nodes, that are so cheap, that they even might be deployed by throwing them from an aircraft in large quantities. Since these devices are battery powered, energy consumption is critical and determines the lifetime of the nodes in the network. Typically most energy is used by routing information through the network, therefore, data aggregation and data fusion are among the most important methods of saving energy. To do this, data needs to be stored until aggregated within the network, but the longer data is stored in the network the higher is the risk of data loss. Furthermore, aggregated data is 'more valuable' than raw data and should be protected according to its value against loss. Therefore, a reliable local storage is needed. Persistent storage like e.g. flash memory is not useful, since a node failure is equivalent to a node loss in sensor networks. Therefore, redundancy is the only way of protecting data against loss.

Of course there needs to be some tuning: HADES was optimized to achieve high speed, whereas in sensor networks the optimization must be done with regard to minimal power and hardware requirements, e.g. by adjusting HADES to use simpler network protocols than TCP and/or by replacing parallel, interleaved operations by serial operations that require less memory.

Another potential application of HADES is for support of mobility in publish/subscribe systems, where events and event histories must be made persistent (Cilia et al., 2003).

References

Akyildiz, I. F., Su, W., Sankarasubramaniam, Y., and Cayirci, E. (2002). A Survey on Sensor Networks. *IEEE Comm. Magazine*, pages 102–114.

Apers, P. M. G., van den Berg, C. A., Flokstra, J., Grefen, P. W. P. J., Kersten, M. L., and Wilschut, A. N. (1992). PRISMA/DB: A parallel main memory relational DBMS. *Knowledge and Data Engineering*, 4(6):541–554.

Carzaniga, A., Rosenblum, D. S., and Wolf, A. L. (2000). Achieving scalability and expressiveness in an Internet-scale event notification service. In *Symposium on Principles of Distributed Computing*, pages 219–227.

Cilia, M., Fiege, L., Haul, C., Zeidler, A., and Buchmann, A. P. (2003). Looking into the past: enhancing mobile publish/subscribe middleware. In *Proceedings of the 2nd international workshop on Distributed event-based systems*, pages 1–8. ACM Press.

DeWitt, D. J., Katz, R. H., Olken, F., Shapiro, L. D., Stonebraker, M. R., and Wood, D. (1984). Implementation Techniques for Main Memory Database Systems. In *Proc. of the ACM SIGMOD Conference*, pages 1–8.

Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. (2003). The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131.

Fetzer, C., Högstedt, K., and Suri, N. (2003). Practical Aspects of Designing an IP Take Over Mechanism. In *Proc. of WORDS*.

Gray, J. (1979). Notes on Data Base Operating Systems. In *Operating Systems: An Advanced Course*, pages 393–481. Springer-Verlag.

Gärtner, F. C. (1999). Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Comput. Surv.*, 31(1):1–26.

Kermarrec, A.-M. (1997). Replication For Efficiency And Fault Tolerance In A DSM System. In *Proc. of PDCS'97 Ninth International Conference on Parallel and Distributed Computing and Systems*.

Litwin, W. and Neimat, M.-A. (1996). High-Availability LH* Schemes with Mirroring. In *Conf. on Cooperative Information Systems*, pages 196–205.

Lorie, R. A. (1977). Physical integrity in a large segmented database. *ACM Trans. Database Syst.*, 2(1):91–104.

Mao, Y., Zhang, Y., Wang, D., and Zheng, W. (2002). LND: a reliable multi-tier storage device in NOW. *SIGOPS Oper. Syst. Rev.*, 36(1):70–80.

Patterson, D. A., Gibson, G., and Katz, R. H. (1988). A case for redundant arrays of inexpensive disks (RAID). In *Proc. of the 1988 ACM SIGMOD international conf. on Management of data*, pages 109–116. ACM Press.

Plank, J. S. and Li, K. (1994). Faster Checkpointing with N+1 Parity. In *24th Annual int. symposium on Fault-Tolerant Computing*, pages 288–297.

Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Schenker, S. (2001). A scalable content-addressable network. In *Proc. of the 2001 conf. on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172. ACM Press.

Rowstron, A. I. T. and Druschel, P. (2001). Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Proc. of the IFIP/ACM Intl. Conference on Distributed Systems Platforms Heidelberg*, pages 329–350. Springer-Verlag.

Skeen, D. (1981). Nonblocking commit protocols. In *Proc. of the 1981 ACM SIGMOD international conf. on Management of data*, pages 133–142. ACM Press.

Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. (2001). Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of ACM SIGCOMM*, pages 149–160. ACM Press.

Zhao, B. Y., Kubiatowicz, J. D., and Joseph, A. D. (2001). Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Technical Report UCB/CSD-01-1141, UC Berkeley.