The Data Management Problem in Post-PC Devices and a Solution

Ramakrishna Gummadi   Randy H. Katz
{ramki, randy}@cs.berkeley.edu
Computer Science Division
University of California, Berkeley
387 Soda Hall #1776
Berkeley, CA 94720-1776

Abstract

The demand for network-enabled limited-footprint mobile devices is increasing rapidly. A central challenge that must be addressed in order to use these next-generation devices effectively is efficient data management – **persistent data** *manipulated or required by applications executing on these computationally and communicationally impoverished devices must be* **consistently managed** *and made* **highly available**. *This data management has traditionally been the responsibility of the OS on which applications execute. In this paper, we extend this conventional OS functionality to include post-pc devices. We propose a novel* **programmatic** *solution to the problem of maintaining high data availability while attaining* **eventual consistency [16]** *in the presence of mobility and disconnected operations, device and network failures, and limited device capabilities. We achieve this by using a combination of a novel proxy architecture, a split request-reply queue based on soft-state principles, and a two-tier update/commit protocol. We also exploit strong object typing to provide application-specific conflict handling in order to attain faster eventual consistency, as well as greater probability of automatic reconciliation.*

I. Introduction and Motivation

We are currently witnessing a widespread use of network-enabled mobile devices such as PDAs, smart phones, hand-held PCs, and other portable communication devices. While these devices are extremely useful because they allow untethered access to data, they are constrained by much smaller computation, storage, and communication capabilities compared to their desktop counterparts. These constraints arise because of the fundamental limitations on form-factors and power consumption, apart from economic factors. Another major problem with these small devices is their inherent unreliability - the wireless network connection (either infrared or RF) that they establish to reach the rest of the world is highly susceptible to outage, they may exhaust their battery power anytime, and the storage components that go into them, such as non-volatile memory chips and flash cards, do not really provide durable storage. In this context, efficient data management on these devices throws up a significant challenge, which the operating systems running on these devices must address effectively.

The unit of data manipulation and storage that is popular on traditional desktops and workstations is a file [22]. Files are organized into directories to provide the notion of a file system, and file system management for applications running on these machines, stand-alone or networked, is one of the primary tasks of an operating system. This concept of a file as an unstructured sequence of bytes of unspecified length that can be accessed randomly (the random access assumption may not be true for a small number of device files) is highly useful for traditional applications, because it is the single lowest common denominator which is directly used by many diverse applications, while individual applications requiring more powerful abstractions can easily structure the data in these files to suit their needs without sacrificing performance or correctness. While data available on local disks is manipulated directly by the native file system and device driver components of an operating system, data residing elsewhere in the network is managed typically by a combination of local clients running inside, or at least tightly coupled with, the operating system kernel, and remote servers that themselves directly invoke the services of the kernel. Moreover, data manipulation operations, such as reads and writes, are typically handled *synchronously*, and in a blocking fashion. Also, there is little distinction made between the nature of storage available on local nodes and that available on remote nodes – both use disks to provide durable storage, and while these disks may differ in some respects, such as throughput and storage capacities, the basic functionality provided is the same. This mode of persistent data management works well in practice for conventional machines and applications because disks and wired network connections are fairly reliable, and normal applications require little more support for persistent storage than a file system (they may, however, require other services, such as network connections and device access for manipulating temporary data).

While this synchronous manipulation of files backed up disks is an acceptable approach in the common case, the advent of small devices that must operate with limited and unreliable persistent storage (such as battery-backed memory cards) in an intermittently connected fashion presents serious impediments to extending the conventional data model to include these emerging devices. This is because of the following factors:

1. **Nature of applications running on post-pc devices**: Most applications running on these devices (such as on-site forms data entry and collection, messaging, calendaring, personal information managing (PIM) applications) deal with structured and *strongly-typed* objects and records, such as forms, e-mail and news articles, schedules, multimedia and http objects, and

not byte arrays and files. In fact, most devices, including PDAs such as Palm, do not even support the notion of files or file systems *per se*. Typically, it is possible to describe the structure of the objects that these applications manipulate in a statically and strongly typed language. This object view of description, query and retrieval, and manipulation of data has important implications with respect to conflict handling for consistency (see Section III).

This object view has another advantage: we can *define* an object as the basic unit (primitive) of data that must be *persistently* and *consistently* managed. This definition is also consistent with the *application data unit* (ADU) concepts [23] and the *integrated layer processing* principles. Thus, we need a persistent data storage and manipulation view that is different from conventional files and file systems. Any operating system supporting persistent data management on the client devices must, therefore, also include support for fine-grained object storage and retrieval. Therefore, in our model, we explicitly deal with objects and types that are defined by applications according to their needs, and invoke application-specific consistency-preserving schemes.

2. **Nature of data access**: Synchronous operation has serious fallouts in the presence of *intermittent connectivity* [17]. The "transparent" mechanisms for network access, such as RPC [24], do not work well in the presence of *partial failure* of the server or the network, and must be handled explicitly and carefully by the operating system or support servers running on the client. We can overcome this problem by taking a completely asynchronous approach to data management. Also, some arrangement, such as hoarding and emulation, must be made to handle *disconnected operations* [18] so as to enable the client to work with a cached copy of available data and make updates anywhere anytime.

3. **Constraints on nature of locally available storage**: Unlike their desktop cousins, post-pc devices only have access to limited and unreliable durable storage facilities (such as battery-backed memories, and, almost certainly, no disks) at their disposal. This places an upper bound on both the amount of data that can safely be committed to stable storage locally, as well as the duration for which such storage can be guaranteed. It is, therefore, important for operating systems managing these devices to evolve capabilities that can effectively deal with these restrictions.

4. **Device proliferation with limitations to reliability**: While the number of devices that a user is likely to use to access data is increasing rapidly, each device taken individually is significantly less reliable than a normal computer. Therefore, an operating system must address the following non-trivial question: how to consistently manage data being simultaneously operated upon by *multiple* unreliable heterogeneous devices.

We conclude, therefore, that operating systems tailored for these small devices need to meet a different set of challenges for managing data effectively. These goals are not met in the present-day operating systems, because of which automatic management of data on these devices is a big problem. The inelegant solution to this problem today is to use a manual approach for managing data – the user must take his or her hand-held PC near a personal computer and synchronize data expressly. This approach clearly has several drawbacks, such as featuring a cumbersome, error-prone, and time-consuming manual mode of operation, imposing limits on the number of devices across which data can be accessed, and constraining mobility because of the assumption that a desktop machine is available to store persistent data as required. *In this paper, we present a novel programmatic approach to the problem of maintaining persistent data consistently across devices.* Our contributions in this paper are as follows:

1. **Propose a novel architecture to the problem of data management**: We propose a novel proxy-based two-tier single-master architecture, with split request-reply queues, that allows devices to manage data efficiently and persistently.

2. **Explore the tradeoffs involved in maintaining strong consistency versus availability in the presence of network outages**: There is a fundamental limitation to building systems that simultaneously offer strong or ACID consistency [25] and high availability. We explore this basic tension between data consistency and availability for the special case of post-pc devices, and examine under what conditions one can build systems that can provide both consistency and availability.

3. **Evolve a programmatic and incremental approach to automate this problem**: We demonstrate how current operating systems can be augmented with a data management service layer that can asynchronously handle disconnected, and intermittently connected modes of operations. This service has the desirable property that it insulates an application developer and a user from having to explicitly manage data, just as the file system interface removes the burden of low-level data management from the programmers, and users of their programs.

4. **Present an object view of data management**: We abandon the conventional file system view for data management because of the reasons presented earlier, and, instead, adopt an object view. We show the advantages of this approach by presenting examples that invoke the fundamental application-level framing (ALF) [23], and soft-state [1] principles. We also study relaxed object consistency models and their implications on application semantics.

The rest of this paper is organized as follows. We discuss our basic assumptions and architecture and rationalize them in Section II. In that section, we also present the advantages of this architecture. In Section III, we present the tension between object consistency and availability, and discuss, with examples, the degrees of consistency up to which objects can be managed by our architecture. We dwell upon our programmatic approach and its advantages in Section IV. Here, we also discuss some optimizations that can be applied to the base case, and implications of ALF and eventual consistency. We discuss the related work in Section V, and conclude in Section VI.

II. System Architecture

Our intention is to design a system that can support the fundamental data requirements of *durability, consistency, availability* efficiently. We defer the problem details of consistency versus availability until Section III. In this Section, we examine how durability and consistency can be achieved.

The growth and deployment of post-pc access devices brings up an interesting question: how can users access their data ubiquitously but safely. To answer this question, one must first answer the more basic question: where exactly does data reside. We claim that emerging technologies [26] [27] indicate that data reposes permanently deep inside the network in data centers that are managed by a federation of commercial utility providers who make this data highly available from anywhere in the network, exploit automatic replication for disaster recovery, employ strong security by default, and provide performance that is similar to that of local storage under normal conditions. Note that this view of storage systems for data is a direct extension of the conventional LAN server storage model, except that it envisages a global-scale ubiquitous data access. Thus, while numerous conventional systems [2] [3] [4] [18] have tackled the problem of providing data access to users in a LAN setting where the nature of access devices, such as laptops, is roughly similar to high-end storage servers, we are now faced with the problem of dealing with data accessed through devices whose nature is very different from the highly sophisticated, redundant, self-repairing, and secure cluster servers in which data resides. This concept of "data utility-providers" is gaining popularity because of the demands of users for data access, the rapid deployment of Internet technologies, and the economies of scale that can be exploited by commercial providers. Thus, our first assumption is that data is managed in *BASEs*, that are scalable, highly available, cluster servers with persistent state, public-key infrastructure, and database support. BASE is also an acronym for Basically Available, Soft-state with Eventual consistency [6], which is the antithesis of ACID [25] semantics for databases. It also accurately highlights the fundamental feature of eventual consistency that is attained at bases based on soft-state principles. We elaborate upon this later in this section.

If we accept this model, the next question is how best to connect these access devices to the network. One choice is for users to establish direct connections to their data repositories. This choice, however, has its limitations: radio wireless connections are notoriously unreliable, slow, power-hungry, and expensive; many devices have other, possibly faster and power-efficient mechanisms for connections that can be used under special conditions, such as wireless infrared connections to line-of-sight networked devices; and, most importantly, the asynchronous and non-blocking requirement of operation does not warrant a continuous connection. Therefore, we propose an *active proxy* approach to connect to bases: active proxies, so called because they are middleware service components with facilities for resource discovery, secure automatic path creation to appropriate bases, object caching, and transformation, are the gateways through which data is accessed by post-pc devices. These proxies include contraptions such as kiosks and online terminals in university buildings, retail locations and public places, and are being increasingly deployed [26] to provide network connectivity to post-pc devices. Proxies have the following advantages:

1. **Client adaptation**: Proxies allow clients that speak diverse network protocols (such as Irda, serial point-to-point, proprietary RF) to adapt to the connected network that speaks TCP/IP. It is unreasonable to expect every access device to speak native TCP/IP because of efficiency considerations; sometimes, as in the infrared case, even if these devices can communicate TCP/IP, they may not be able to establish direct wide-area connections. Also, the high round-trip times and loss rates of wireless connections make TCP/IP implementations bloated, unwieldy, and inefficient [7]. Besides, proxies directly support mobility and disconnected operations because objects downloaded from one proxy may be updated at the data repository through another proxy after any length of disconnected operation.
2. **Efficient resource management**: Proxies, where available, can serve to multiplex network connections, offering high-bandwidth wireless network connections. Scarce resources, such as IP addresses, may be managed efficiently through a proxy approach [7].
3. **Automatic content transformation**: Through mechanisms such as object transformation, aggregation, caching, and customization (TACC) [6], it is possible to optimize data management performance according to application requirements and user preferences.

The capabilities of proxies, such as resource discovery, secure communication with bases, and TACC capabilities, can be captured in terms of *operators* that can be *composed* under the generic name of *paths* [17]. While proxies are well-connected to the infrastructure, they are shared, untrusted resources that are not particularly reliable, and, therefore, do not support persistent data or high availability. We have seen how to achieve the former using bases; the latter can be achieved by using a collection of proxies that depend only on soft-state [1]. Finally, bases themselves can contain proxies as middleware components that can be accessed by users when they have a direct connection to bases, but not to proxies.

A principal feature of our system architecture is a two-tier object update/commit protocol. The two-tiers correspond to a first-level tentative update from access devices to proxies, and a final second-level update from proxies to bases. Both the updates are asynchronous with respect to each other. This is roughly analogous to the two-tier model described in [20], except that we do *not* hold locks around updates, and, instead, use optimistic forms of concurrency [1] and consistency control, primarily for high availability.

The overall architecture is depicted in Figure 1, which shows how end-devices connect to bases through proxies. The request queue stores objects that need to be checked in to the base, and is managed by an operating system service running on the client. The reply queue contains objects that are downloaded from the base for servicing client cache misses and client commit confirmation replies for previously checked-in objects that have been successfully merged at the base, and is managed by the proxy. The reply queue contains entries whose state is soft – entries are retired from the queue either when a client dequeues them, or after a timer expires. It is acceptable for the state to be soft because there is still a persistent state at the base, which can be accessed by the client, either through the same, or a different proxy, after a lengthy disconnection. Also, because of this flexibility (arising due to proxies operating in a soft-state mode) in selecting a proxy, we can guarantee high availability of proxies.

Given this model, we can define object persistence as follows: we consider an object to have write-stabilized [16] if and only if a copy of the object reaches its base, where, after passing object-

[1] for updates made to the same object across different proxies

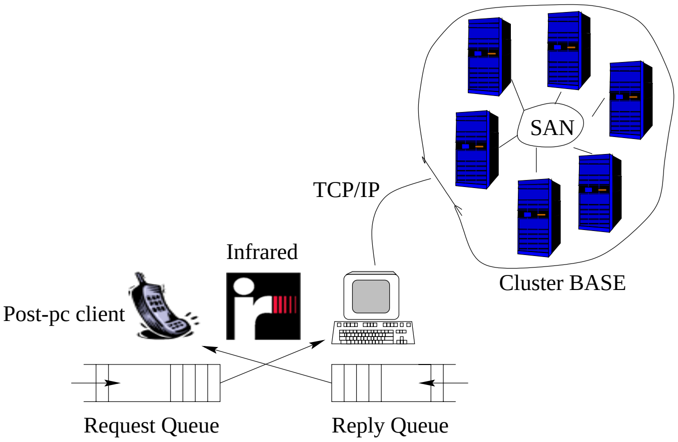

Fig. 1. Overview of the Data Management Service Architecture

specific per-write dependency checks, it is merged with other objects and committed to stable storage. The copies that are manipulated by the post-pc units are only leased (can be recalled if inconsistencies show up) and cached copies of objects that are stored immutably in bases.

Thus, access devices may manipulate objects that are downloaded through an active proxy, and may check in modified copies of these objects through another proxy. Enough log information is generated by our operating system service running on the access devices so that the updates propagated to bases through proxies can be unambiguously ordered and merged at the bases. This procedure is described in more detail in Section IV. So, object consistency is defined from the perspective of the write-order seen at the base.

In summary, we employ a proxy-brokered, infrastructure-backed, single master data management scheme for consistent, highly available, and persistent storage management. These ideas are summarized in Table I [17].

Type	Nature	State	Connectivity	Capability
Clients	End devices	Cache only	Zero or meager	Talk to proxies
Active Proxies	Shared, untrusted	Soft state	Good, talk with bases	+Operators, paths
Bases	Trusted, Well-run	Master durable	No partitions	+Highly available

TABLE I

III. Degrees of Object Consistency versus Availability

There is a fundamental theorem [17] governing data availability, consistency, and network characteristics, called the **CAP** theorem, which we have exploited for supporting efficient data management. It states that it is only possible to choose two of the three features of strong consistency, availability, and network partitioning. One **can not** afford the luxury of working under all three conditions - either we forego strong consistency for maintaining high data availability under network partitions, or we forego high availability for maintaining strong consistency in the presence of network partitions by making one side of a network partition unavailable (typically, the smaller side), or we create special conditions for having both strong consistency and high availability (by ensuring no network partitioning). Traditionally, databases have adopted the first approach, making them unscalable for handling large number of simultaneous network connections from clients by techniques such as redundancy and replication, while all weakly consistent systems, such as Bayou [16] and Coda [18], have taken the second path. The third case is familiar in single node desktop machines, where there is no scope for partial failures.

We strive to attain consistency and high availability in bases by using cluster computers interconnected by high-speed system-area networks (SANs) running in highly controlled environments that have a very low probability of failure (the SAN, for example, can be constructed to have a cyclic topology so that the network does not partition under a single node or router failure [17]) so as to scale across the number of wide-area client requests that can be processed while presenting a single-master view of data outside the base (because multiple master replicas have a fair share of problems, such as slow convergence, and greater probability of conflicts [19] [16]). We attain high availability by replicating persistent objects inside bases so that we can tolerate *independent failures* of components. We insure degree 3 consistency [25] among the replicas through a lazy update protocol [8] [12]. Thereby, we attain both strong consistency and high availability within a base.

Since the connections between a user and the proxies, as well as between proxies and bases may partition, we can not hope to have strong consistency if we simultaneously want high availability. We therefore relax the strong (degree 3) consistency requirement, opting, instead, for eventual consistency, and, by applying application-specific conflict resolution procedures, for low probability of requiring manual reconciliation of conflicts. This probability can be lowered further by electing lower degrees of consistency, such as degree 2 or degree 1, without drastically affecting the application semantics: for example, we may risk e-mail messages that are treated as objects getting reordered while a user's mailbox is updated at the base if we use use two different transactions with degree 2 consistency for this purpose instead of a single transaction with degree 3 consistency; but this is not a serious problem (if the client really cares, the email reader at the client can include identifiers inside messages and present the messages in the correct order finally to the user). On the other hand, we gain flexibility and parallelism by using two transactions because the two different transactions can be simultaneously initiated by two different proxies, and serviced by two different replicas within a base. Thus, in the email example, consider a transaction T2 that read the mailbox length that has been modified (but not yet committed) by transaction T1 which had started earlier, and updated the mailbox length to account for the new message; if T1 were to now abort for some reason, such as the replica on which T1 was running has crashed, etc., T2 would also have to abort, or else the mailbox would be left in an inconsistent state. We avoid such problems by carefully choosing the correct degree of consistency required for an application, and enforcing it efficiently and correctly within the cluster. Single machine Unix systems do not encounter such problems because server processes are carefully multi-threaded to avoid problems on single machines, but this is still a limitation of the system because such processes can not work together efficiently as a cluster, although it is fairly easy to make them work correctly by using coarse-grained file locks. Note that Unix file systems also do not support avoiding *Read/Write* conflicts, required for degree 2 consistency, because Unix does not have the notion of atomicity beyond the boundary of a system call, and this is another limitation in developing efficient distributed applications.

IV. The Programmatic Approach

We now examine the major components of our data management service that need to be present on the end-device clients, active proxies, and bases, and how they interact with each other. In our proposed implementation, we have an *object manager* running on the mobile unit that manages the limited persistent storage available on the devices intelligently by multiplexing persistent storage amongst uncommitted, but modified, objects belonging to all applications. Applications are required to type objects strongly, so that application-specific conflict resolution policies can be initiated at the bases. This object manager also doubles as a cache and hoard manager, automatically detecting the hot-set of objects that need to be kept around in memory, and accepting requests from applications about objects that need to be retained or prefetched. Thus, it possible for applications on the clients to operate completely asynchronously. It also holds a log of all updates made to objects from the last time of successful commit. When disconnected, it serves objects from its cache if it can, and records any update activity. When connected to a proxy, it initiates a secure connection after appropriate authentication, and downloads the set of modified objects and the update log into the proxy. To economize on log size, we opt for logical logging [9], and the log records are retained until explicit commit confirmations for the corresponding items are received from bases through the response queues in the proxies (see Figure 1), although the objects themselves may be purged earlier. The idea is that once the items are downloaded into the proxies, they are stored until a connection can be established with the base. Thus, proxies serve to decouple the user from a base, and users interact with bases by means of the request queue present on the end-device, and the response queue present on the proxy (see Figure 1).

The approach of manipulating Application Data Units [23] has been widely recognized to be a good design principle, and, in our case, the unit of manipulation at the base is an object, such as a form, a database tuple, calendar entry, or an email message, that can be specified according to the needs of an application. An entire object is updated atomically at a base once all modifications of an object are shipped from a proxy to the base. For performance reasons, multiple objects can be simultaneously transferred from a proxy to the base, where each one is handled individually. Multiple proxies may simultaneously apply changes to an object at a base (this can happen, for example, due to user mobility), but each update is rendered atomic locally at the base, and *no locks* are required between a proxy and a base. In effect, we achieve update serialization across proxies by performing all updates locally at the bases. Local fine-grained locking techniques would have to be used within a base, but locks can be acquired and released quickly, and deadlocks can be detected fairly cheaply; it is also possible to use lock-free implementations [21].

There is one problem with this approach: if the master server running on the base does not know the list of proxies that contain copies of objects checked in, how can we achieve eventual consistency? The answer is that the client carries log information about the objects that have been submitted through various proxies for re-integration at the bases. By our assumption that a proxy is well-connected, we can guarantee that eventually, these updates can reach the master server. In this scheme, we do not need expensive pairwise reconciliation between proxies. The log records about objects checked in through various proxies can be purged at the client once the server assures it that the object updates have successfully retired. If there were any object conflicts, they explicitly show up, just like in the popular concurrent versions system (CVS); but this is the rare case, and the conflicts can be remedied fairly easily manually.

V. Related Work

A number of systems for providing high data availability in conventional systems have been built, and these include systems such as Bayou [16], Clearinghouse [14], Coda [18], Rover [13], CVS [28], Ficus [10], Grapevine [2], and Lotus Notes [15]. We exploit several of their properties, such as weak consistency, replay logs, and version vectors. We also differ from them in several respects: explicit use of objects manipulated by post-pc devices and managed persistently and consistently by single masters, provision for different degrees of consistency within transactions occurring in bases, and partitioning of functionality among end-units, proxies, and bases.

The Bayou project at Xerox PARC extensively studied the challenges involved in building weakly consistent replicated servers for disconnected operations. It uses version vectors developed in the context of the Locus project at UCLA. Grapevine was one of the earliest weakly replicated systems that propagated updates via electronic mail. This was extended to use a background anti-entropy process in Clearinghouse. Pair-wise reconciliation of replicas is currently used in several systems, such as Lotus Notes, and Ficus. Systems that organize replicas into hierarchies, where a replica only exchanges updates with its parents or children, have also been explored. Examples include client-server reconciliation protocols in file systems like Coda, and distributed object systems like Rover. Database systems, such as Oracle [11] and Sybase [5], use a primary-secondary or master-snapshot protocols for data management. Because communication patterns are simple, these systems can easily maintain accurate information about the state of the replicas with which they exchange updates. Rover uses asynchronous queued RPC invocations that are eventually applied to the master copy of an object. More recently, the Oceanstore project [15] at UC Berkeley is currently studying the problem of providing ubiquitous, highly available, reliable, and persistent storage on the global scale. In the commercial world, companies such as Streetspace [26] are setting up terminals, that are similar to proxies, in public places and retail establishments, while companies such as Fusionone [27] are providing software that make information access seamless across multiple computing devices. The problem with the latter approach is that the user has to be explicitly involved in retrieving relevant data and synchronizing it back into the network, and there is no programmatic service that applications can leverage.

While many systems mentioned above are designed to support a file system view, we believe that post-pc operating systems should support an object view of data management, both because the probability of automatic conflict resolution is greater, and because eventual consistency is more swiftly attained: for example, in Coda, an unresolvable situation arises much less frequently for directories than for normal files because directly are well-structured.

VI. Conclusions and Future Work

In this paper, we have described and rationalized an architecture for data management for small devices. We have also shown how an operating system can provide this functionality through an additional service layer that every application running on the end-unit can make use of. This scheme makes it incrementally deployable, because new applications can be developed to take direct advantage of this service, while existing applications can use a wrapper that services requests for objects which applications assume are stored persistently within the device. We believe the architecture also reflects the trends in the deployment of commercial deployment of Internet technologies, where cluster computing platforms and proxies are increasingly gaining in importance as service deployment platforms.

We are currently implementing the data management layer in Java for extensibility, and portability reasons. This service can then run inside a tiny JVM, such as Waba [31] or KVM [29]. We leverage the work done in projects such as Bayou and Coda to provide a programmatic scalable, and available solution that guarantees eventual consistency. A key requirement in our architecture is security: all communication between the client devices and proxies, as well as between proxies and bases must be strongly authenticated and encrypted. To this end, we are developing a secure RMI layer within the JVM so that secure RMI connections can be initiated between clients and proxies. We are also actively examining the more difficult problem of treating the proxies themselves as completely untrustable infrastructure services.

In conclusion, in this paper, we have examined the problem of operating system support for efficient data management in small devices, proposed a new architecture based on current possibilities and trends, dealt explicitly with objects managed by single masters which are made both highly available and strongly consistent, and demonstrated how a single middleware layer of proxies can provide efficient asynchronous data support for storage constrained mobile devices.

References

[1] Amir, E., McCanne, S., and Katz, R. An Active Service Framework and it Application to Real-time Multimedia Transcoding. *Proceedings of SIGCOMM 1998*. Vancouver, Canada, Sep 1998.

[2] Birrell, A., Levin, R., Needham, R. M., and Schroeder, M. D. Grapevine: An exercise in distributed computing. *Communications of the ACM*. Apr 1982.

[3] Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., and Terry, D.. Epidemic algorithms for replicated database maintenance. *Proceedings Sixth Symposium on Principles of Distributed Computing*. Vancouver, Canada, Aug 1987.

[4] Golding, R. A. A weak-consistency architecture for distributed information services. *Computing Systems, 5(4):379-405*, Fall 1992.

[5] Gorelik, A., Wang, Y., and Deppe, M. Sybase Replication Server. *Proceedings of the 1994 ACM SIGMOD Conference*. Minneapolis, Minnesota, May 1994.

[6] Fox, A., Gribble, S. D., Chawathe, Y., Brewer, E. A., and Gauthier, P. Cluster-Based Scalable Network Services. *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*. Saint-Maolo, France, Oct 1997.

[7] Balakrishnan, H. Challenges to Reliable Data Transport over Heterogenous Wireless Networks. *Ph.D. thesis, UC Berkeley*. Aug 1998.

[8] Gribble, S. Simplifying Cluster-Based Internet Service Construction with Scalable Distributed Data Structures. *Ph.D. Candidacy Qualifying Exam*. Apr 1999.

[9] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*. 17(1): 94-162, 1992.

[10] Guy, R. G., Heidemann, J. S., Mak, W., Page, T. W., Popek, G. J., Rothmeier, D. Implementation of the Ficus replicated file system. *Proceedings of Summer USENIX Conference*. June 1990.

[11] Oracle Corporation. Oracle7 Server Distributed Systems: Replicated Data, Release 7.1 . Part No. A21903-2, 1995.

[12] Ladin, R., Liskov, B., Shrira, L., and Ghemawat, S. Providing high availability using lazy replication. *ACM Transactions on Computer Systems*. 10(4), November, 1992.

[13] Joseph, A. D., deLespinasse, A. F., Tauber, J. A., Gifford, D. A., and Kaashoek, M. F. Rover: A toolkit for mobile information access. *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. Copper Mountain, Colorado, Dec 1995.

[14] Oppen, D. C., and Dalal, Y. K. The Clearinghouse: A decentralized agent for locating named objects in a distributed environment. *ACM Transactions on Office Information Systems*. 1(3), July 1983.

[15] Kalwell Jr., L, Beckhardt, S., Halvorsen, T., Ozzie, R., and Grief, I. Replicated document management in a group communication system. *Groupware Software for Computer-Supported Cooperative Work*. Ed. by D. Marca and G. Bock, IEEE Computer Society Press, 1992.

[16] Peterson, K., Spreitzer, M. J., Terry, D. B., Theimer, M. M., and Demers, A. J. Flexible Update Propagation for Weakly Consistent Replication. *Proceedings of the 16th ACM Symposium on Operating Systems Principles*. Saint-Malo, France, Oct 1997.

[17] Brewer, A. B., The Ninja Architecture for Robust Distributed Systems. Unpublished manuscript, UC Berkeley, Jan 2000.

[18] Kistler, J. J., and Satyanarayanan, M. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*. Vol. 10, No. 1., Feb 1992.

[19] Gray, J. N., Lorie, R. A., Putzolu, G. R., and Traiger, I. L. Granularity of Locks and Degrees of Consistency in a Shared Data Base. *IFIP Working Conference on Modeling on Data Base Management Systems*. AFIPS Press, 1977.

[20] Gray, J., Helland, P., O' Neil, P., and Shasha, D. The Dangers of Replication and a Solution. *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*. pp. 173-182, 1996.

[21] Greenwald, M., and Cheriton, D. R. The Synergy Between Non-blocking Synchronization and Operating System Structure. *Proceedings of the Second Symposium on Operating System Design and Implementation*. USENIX, Seattle, October, 1996.

[22] Ritchie, D. M., and Thompson, K. The UNIX Time-Sharing System. *Proceedings of the Fourth ACM Symposium on Operating Systems Principles*. Yorktown Heights, New York, Oct 1973.

[23] Clark, D., and Tennenhouse, D. Architectural Consideration for a New Generation of Protocols. *Proceedings of SIGCOMM 1990*. Philadelphia, PA, Sep 1990.

[24] Srinivasan, R. RPC: Remote Procedure Call Protocol Specification Version 2. *RFC 1831*, Aug 1995.

[25] Gray, J., and Reuter, A. Transaction Processing: Concepts and Techniques. *Morgan Kaufman Publishers*. San Francisco, 1993.

[26] http://www.streetspace.com/

[27] http://www.fusionone.com/

[28] http://www.loria.fr/ molli/cvs/doc/cvs_ toc.html

[29] http://java.sun.com/products/kvm/

[30] http://oceanstore.cs.berkeley.edu/

[31] http://www.wabasoft.com/