Towards a storage system for connected homes

Trinabh Gupta (*UT Austin*) Amar Phanishayee Jaeyeon Jung Ratul Mahajan

Microsoft Research

1 Introduction

Homes are increasingly filled with connected devices such as wireless door locks, remotely controllable thermostats, and security cameras. Home automation systems, which used to be prohibitively expensive to average consumers, are now offered at an affordable price, accelerating the adoption of "smart" devices in the home. The advances in devices also enable researchers and practitioners to develop new services and applications for householders—PreHeat uses occupancy sensing to efficiently heat homes [19]; a DigiSwitch display supports elders who reside separately from their caregivers with whom they share sensed activity data in the home [5]; and a Digital Neighbourhood Watch (DNW) is proposed to help neighbors detect suspicious activities (e.g., previously unseen cars driving by several times) in the neighborhood by sharing security camera images [4].

To facilitate such home technology developments, platforms like HomeOS [6] and Mi Casa Verde [1] provide a standard way to communicate with a range of devices and a set of APIs to ease the implementation of applications. However, storage abstraction for applications in the home is not yet clearly defined. For instance, HomeOS exposes local file system APIs to HomeOS applications, but has no built-in support for sharing data with other applications that may run remotely. Commercial devices typically come with a tailored storage solution (e.g., *Withings* wireless scales transmit data to `withings.com` by default) and as a result, data from each device is stored in silos, causing a data management nightmare.

We first discuss the key requirements for a storage system that supports connected devices and applications in the home. We ground our discussion with the use cases derived from the aforementioned examples—PreHeat, DigiSwitch, and DNW—as well as our own experience of building new applications using various in-home sensors. We then present Bolt, a storage system for data generated by connected devices and applications in the home. Bolt offers a stream based key-value abstraction with support for range queries over time and filtering based on application-specific keys. Bolt stores data on potentially untrusted cloud storage providers while ensuring confidentiality using decentralized access control. Storage policies in Bolt allow applications to prioritize their storage requirements of space, performance, cost, and reliability. After highlighting the key difference between Bolt and prior systems, we finally conclude the paper with a description of our ongoing efforts to build Bolt.

2 Storage Requirements

Optimize storage & retrieval for time series data: Many devices in the home generate continuous time series data (e.g., security video data) although some devices generate data on demand (e.g., wireless scale) or infrequently (e.g., motion-sensors). Hence, a storage system for these devices should support common operations over time series data such as temporal range queries. Another important observation is that time series data generated by applications and connected devices at home have *single* writers simplifying concurrency control and consistency protocols. Writers always generate new data and do not perform random-access updates or deletes. Traditional databases with their support for transactions, concurrency control, and recovery protocols are an overkill for such data as also noted in [20], and file based storage offers inade-

quate query interfaces. Systems that store time series data should support efficient querying of massive datasets while avoiding the overheads in supporting generic storage workloads.

Support efficient sharing of data across devices: Data generated by devices should be easily shareable with applications no matter where the applications are running. For example, the PreHeat application needs to access data generated by a thermostat and occupancy sensors running in a single home whereas the DNW application needs to access data generated by devices managed by different homes. Applications may want to access only part of the data produced by a device. For example, in the DNW example, it would be wasteful to access the entire day worth of video data if the search for suspicious activities needs to be done only over the past few hours. This raises an important question: what should the sharing granularity be? The sharing granularity also affects the efficiency of reads and writes; these frequent operations should incur a low overhead in terms of CPU cycles, storage overhead, and communication bandwidth and latency.

Ensure confidentiality of data: Data generated by devices in the home may contain sensitive information and therefore the home storage system must guarantee the confidentiality of stored data and enforce access control specified by the data owner. Previous studies (e.g., [16]) show that unlike enterprise settings, access control for home data sharing needs to support semantic grouping (based on the content of data) and flexible policies as users' ideal policies are nuanced, complex, and have many exceptions. Hence, open research questions include efficiently supporting changes in access controls without requiring frequent re-encryption of a large amount of data and providing granular access control capabilities with an easy-to-use policy authoring interface.

Support policy-driven storage: Data generated by devices and applications have different storage requirements for access performance, cost, and reliability. For example, a camera that records images upon detecting motion might store them locally at home and delete them once the DNW application has extracted images with people in them. The DNW application might store these selected images on a remote server to correlate individuals in images captured by neighbouring cameras; once analyzed, they can be stored on cheaper, slower archival storage servers. Applications are in the best position to prioritize storage metrics by specifying appropriate policies.

As we review in detail in Section 4, existing storage systems either expose inefficient sharing and query abstractions for temporal device data [7, 9, 13], assume partial or complete trust on the storage servers [14], or store data locally within the home while ignoring application specific storage policies [10]. Achieving the above four requirements simultaneously is challenging as they are at odds with each other. For example, storing data locally facilitates privacy but it inhibits sharing, remote access, and reliable storage. By the same token, storing data on external servers in the cloud provides reliable storage and enables easy remote access & sharing, but untrusted storage servers can violate privacy. Finally, naïvely storing encrypted data on untrusted servers inhibits efficient sharing of data.

3 Initial Prototype of Bolt

Motivated by the requirements listed in Section 2, we first discuss three guiding design decisions, followed by the design of our initial Bolt prototype.

3.1 Design Decisions

Key-value streams as an abstraction for time-series data: Bolt abstracts data into *streams* which are uniquely identified by the <HomeID, AppID, StreamID> tuple. A stream is made up of data blocks and indices to support efficient lookups using time and application-specific keys. The granularity of sharing in Bolt is a stream.

Storage servers not trusted for confidentiality or integrity: Bolt uses decentralized access control to ensure the confidentiality of data; it encrypts the data blocks of the stream and distributes keys to readers with the help of a trusted key server. Bolt uses key regression [8] and lazy revocation [12] for efficient distribution of keys. Encryption provides confidentiality of data, however, a remote untrusted server storing part of a stream may modify data blocks or even return old data. Bolt guarantees the detection of data integrity violation and con-

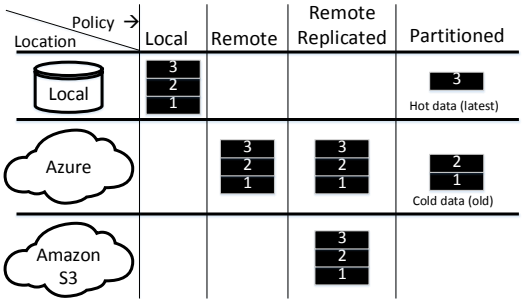

Figure 1: Policies specify the storage location for stream data (here, data blocks 1, 2, and 3). Bolt currently supports local, Azure, and S3 storage.

Function	Description
append(key, val)	append data for a key
update(key, val)	update latest value for a key
getLatest()	get latest KV pair inserted
get(key)	get latest value for key
getAll(key)	get all time-sorted values for key
getAll(key, t_s, t_e)	get all values for key in time interval
getAll(k_{start}, k_{end})	get all keys in key range
grant(appid)	grant appid read access
revoke(appid)	revoke appid's read access
delete()	delete stream data

Figure 2: Bolt's stream APIs for storage, retrieval, and sharing.

tent freshness. For efficient reads and writes, once a stream is opened all data transfers occur only between the storage server and the application, minimizing dependency on the key server. Integrity and freshness checks are performed initially when a stream is opened for read or write; the integrity of individual data blocks is checked on reads.

Configurable streams use storage across different providers: The location of a stream is configurable (Figure 1). It may be stored either locally, remotely on untrusted servers, replicated for reliability, or striped across multiple storage providers for cost effectiveness. This configurability allow users and applications to prioritize their storage requirements of space, performance, cost, and reliability. Stream data is stored in a log enabling efficient uploads on write. An index makes reads efficient, fetching only the required data from local or remote storage.

3.2 Bolt Prototype

API and guarantees. Bolt provides applications two type of streams: (i) a `ValueStream` for small data values (e.g. temperature readings); and (ii) a `FileStream` for large values (e.g. images, videos). ValueStreams append all stream data to a single file and FileStreams store each entry in a separate file. Applications specify `policies` corresponding to those shown in Figure 1. Each stream can have one owner application (writer) and many readers. Figure 2 shows the Bolt stream APIs. A writer can store data values and tag them with a key using `append` & `update`, and it can `grant` and `revoke` read access to data. Readers can filter and query data using application-specific keys and time (`get*`).

Bolt provides three security and privacy guarantees: (i) Confidentiality: Data written to a stream can only be read by an application to which the owner grants access, and once the owner revokes a reader's access to a stream, the reader cannot access data generated *after revocation*; (ii) Tamper evidence: readers can detect if data has been tampered with by anyone other than the owner; (iii) Freshness: readers can detect if the storage server returns stale data. Bolt does not defend against denial-of-service from the untrusted storage.

Read, write, and access control. Each principal (<HomeID, AppID> tuple) in Bolt is associated with a private-public key pair. A key server maintains the principal to public-key mappings. Each stream is associated with metadata: a symmetric content key to encrypt and decrypt data (K_{con}), key version, principals that have access to the data (one of them is the owner), and the location where stream data is stored. K_{con} is stored encrypted — one entry for each principal that has access to the stream data using their public key. Stream data consists of two parts: a log of encrypted data blocks, and an index that maps a tag (key) to a time-sorted list of data block identifiers.

Figure 3 shows the steps during reads and writes in Bolt. An owner opens a stream (step 1) and

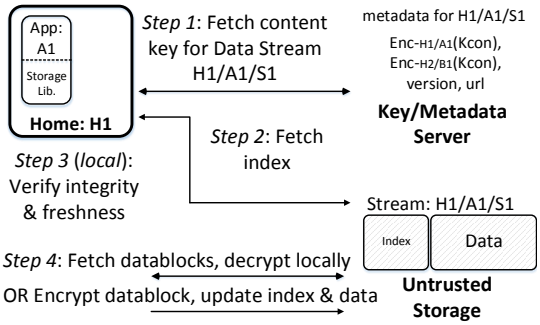

Figure 3: Steps during reads and writes for application A1 in home H1 accessing stream $H1/A1/S1$.

fetches stream metadata. [1] Next the owner fetches the stream's index from the untrusted storage server (step 2). The index is made up of two parts: (i) index data that maps a key to a list of data block identifiers (<data-log-offset, write-timestamp, hash(data block)> tuples), and (ii) index metadata which contains the owner signed hash of index data. On verifying the integrity of the index data using index metadata (step 3), the owner stores data blocks encrypted with the content key (<key, value, write-timestamp, key-version> tuples); data blocks and the modified index are stored back on the server (step 4). Reads proceed similarly, with the integrity of read data blocks being verified by using index data.

To grant applications read access, the owner updates stream metadata with the content key encrypted with the reader's public key. Revoking read access also involves updating stream metadata: owners remove the appropriate principal from the accessor's list, remove the encrypted content keys, roll forward the content key and key version for all valid principals as per key-regression. Key regression allows readers with version V of the key to generate keys for versions 0 to $V-1$.

Bolt provides guarantees on data freshness similar to SFSRO and Chefs [9]. Bolt clients check content freshness using time stamps, ensuring any data fetched from a storage server is no older than a configurable writer-specified consistency period, and also no older than any previously retrieved data.

[1] In our initial design, we assume that stream metadata is stored on a trusted key server to prevent unauthorized updates.

4 Related Work

A number of recent works focus on building systems for sharing and managing personal data. We first highlight key differences between Bolt and several closely related systems. We then discuss prior studies on securing data stored in untrusted remote storage and how they compare to Bolt.

Personal and home data management: Perspective [18] is a semantic file system designed to help householders easily manage data spread across devices (e.g., portable music player, DVR, laptop) in the home. Perspective exposes a *view* abstraction where a view is an attribute-based description of a set of files with support for queries on file attributes. It allows devices to participate in the system in a peer-to-peer fashion. Security and access control are not a focus of the work. HomeViews [10] is designed to ease the management and sharing of files among people. It exposes database-style views over one's files and supports access-controlled sharing of views with remote users in a peer-to-peer based architecture. Both systems are more suited for storing, managing, and sharing user generated data (e.g., photos, digital music, documents) rather than device-generated time series data.

Secure systems using untrusted storage: SUNDR [13] is a network file system that provides integrity and consistency guarantees of files stored in untrusted remote storage. SPORC [7] is a framework for building group collaboration services like shared documents using untrusted servers. Venus [21] and Depot [15] expose a key-value store to clients on top of untrusted cloud storage providers. Chefs [9] enables replicating an entire file system on untrusted storage servers in order to support a large number of readers. All of these systems expose a storage interface on top of untrusted storage, however, none is suited for supporting semi-structured time series data from connected devices. These systems also do not provide configurability on where to store data: local versus remote for privacy concerns, partitioned across multiple storage providers for cost-effectiveness, and replicated across multiple providers for reliability and avoiding vendor lock-in (as in RACS [2] and HAIL [3]). A related strand of work focuses on accountability and auditing

	Sharing granularity	*Access control models*	*Datastore trust assumptions*
Perspective [18]	Views (file attributes)	N/A	Trusted devices owned by the user
HomeViews [10]	Database-style views	Capability-based access control per view	Trusted local database that communicates with others in a P2P fashion
SUNDR [13]	Individual files	File owned by a user or group; no support for read access control	Untrusted remote storage (mainly focused on data integrity)
SiRiUS [11]	Individual files	Read & write access control per file	Untrusted remote storage (focused both on data integrity and secrecy)
Chefs [9]	Entire file system	Read-only access control for entire file system	Replicated untrusted remote storage (focused both on data integrity and secrecy)
Bolt	Individual streams with <time, key, value> tuples	Read-only access control per stream	Trusted local and untrusted, replicated, or partitioned remote storage (focused on integrity and secrecy)

Figure 4: Comparing Bolt with prior systems in terms of (1) data sharing granularity, (2) access control models, and (3) trust assumptions that each system makes on data storage.

(see Cloudproof [17]) of cloud behavior but again they are not suitable for the home setting and require server-side changes. Ming et al. [14] store patient health records (PHR) on the cloud and enable attribute-based access control policies to enable secure and efficient sharing of PHR's. However, their system again requires cooperation from storage servers. Goh et al. [11] propose a security overlay called SiRiUS that extends local file systems with untrusted cloud storage systems with the support of data integrity and confidentiality. SiRiUS supports multiple writers and readers per file but does not provide freshness guarantee of data content. Figure 4 compares Bolt with some of the aforementioned systems.

5 Future Directions

Our Bolt prototype currently supports local, Azure, and S3 storage, and we have integrated Bolt with HomeOS. We are currently working on several extensions. Our current design trusts the key/metadata server to prevent unauthorized updates to metadata. Moving forward, we are looking at ways to minimize this trust. One approach is to replicate metadata at $2f+1$ servers and go by majority, to tolerate up to f malicious servers. We are also looking at ways to make the sharing granularity more flexible. Currently, a reader gets access to all the data written to a stream. We are looking at efficient ways to share data generated only within a particular time interval or only filtered by a particular key. Approaches to support these schemes will potentially need more metadata storage space; a tradeoff we are exploring next.

References

[1] Mi Casa Verde. http://www.micasaverde.com/. Retrieved: Sept 2013.

[2] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon. Racs: a case for cloud storage diversity. In *SoCC*, 2010.

[3] K. D. Bowers, A. Juels, and A. Oprea. Hail: a high-availability and integrity layer for cloud storage. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Nov. 2009.

[4] A. B. Brush, J. Jung, R. Mahajan, and F. Martinez. Digital Neighborhood Watch: Investigating the Sharing of Camera Data amongst Neighbors. In *CSCW*, 2013.

[5] K. E. Caine, C. Y. Zimmerman, Z. Schall-Zimmerman, W. R. Hazlewood, L. J. Camp, K. H. Connelly, L. L. Huber, and K. Shankar. DigiSwitch: A device to allow older adults to monitor and direct the collection and transmis-

sion of health information collected at home. *J. Medical Systems*, 35(5):1181–1195, 2011.

[6] C. Dixon, R. Mahajan, S. Agarwal, A. J. Brush, B. Lee, S. Saroiu, and P. Bahl. An operating system for the home. In *Proc. 9th USENIX NSDI*, Apr. 2012.

[7] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. SPORC: group collaboration using untrusted cloud resources. In *Proc. 9th USENIX OSDI*, Oct. 2010.

[8] K. Fu. Key regression: Enabling efficient key distribution for secure distributed storage. In *NDSS*, 2006.

[9] K. Fu. *Integrity and access control in untrusted content distribution networks*. PhD thesis, MIT, 2005.

[10] R. Geambasu, M. Balazinska, S. D. Gribble, and H. M. Levy. Homeviews: peer-to-peer middleware for personal data sharing applications. In *SIGMOD*, 2007.

[11] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: Securing Remote Untrusted Storage. In *NDSS*, 2003.

[12] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *Proc. 2nd USENIX Conference on File and Storage Technologies*, Mar. 2003.

[13] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proc. 6th USENIX OSDI*, Dec. 2004.

[14] M. Li, S. Yu, K. Ren, and W. Lou. Securing personal health records in cloud computing: Patient-centric and fine-grained data access control in multi-owner settings. In *SecureComm*, LNICST. Springer Berlin Heidelberg, 2010.

[15] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. *TOCS*, 29 (4), Dec. 2011.

[16] M. L. Mazurek, J. P. Arsenault, J. Bresee, N. Gupta, I. Ion, C. Johns, D. Lee, Y. Liang, J. Olsen, B. Salmon, R. Shay, K. Vaniea, L. Bauer, L. F. Cranor, G. R. Ganger, and M. K. Reiter. Access control for home data sharing: Attitudes, needs and practices. In *Proceedings of the SIGCHI conference on Human factors in computing systems (CHI '10)*, 2010.

[17] R. A. Popa, J. R. Lorch, D. Molnar, H. J. Wang, and L. Zhuang. Enabling security in cloud storage SLAs with CloudProof. In *USENIX ATC*, 2011.

[18] B. Salmon, S. W. Schlosser, L. F. Cranor, and G. R. Ganger. Perspective: semantic data management for the home. In *Proc. USENIX Conference on File and Storage Technologies (FAST)*, Feb. 2008.

[19] J. Scott, A. J. B. Brush, J. Krumm, B. Meyers, M. Hazas, S. Hodges, and N. Villar. PreHeat: controlling home heating using occupancy prediction. In *Ubicomp*, 2011.

[20] I. Shafer, R. R. Sambasivan, A. Rowe, and G. R. Ganger. Specialized storage for big time series. In *The 5th USENIX Workshop on Hot Topics in Storage and File Systems*, HotStorage, 2013.

[21] A. Shraer, C. Cachin, A. Cidon, I. Keidar, Y. Michalevsky, and D. Shaket. Venus: verification for untrusted cloud storage. In *CCSW*, 2010.