SubScribe: Secure and Efficient Data Delivery/Access Services in a Push-Based Environment

Anindya Datta
DuPree College of Management, Georgia Tech.
755 Ferst Drive, Atlanta, GA 30332-0520
email: adatta@cc.gatech.edu

Aslihan Celik
MIS Dept.,University of Arizona, Tucson, AZ 85721
email: aslihan@loochi.bpa.arizona.edu

Rebecca N. Wright
AT&T Labs—Research, 180 Park Avenue, Florham Park, NJ 07932, email: rwright@research.att.com

Alexandros Biliris
AT&T Labs—Research, 180 Park Avenue, Florham Park, NJ 07932, email: biliris@research.att.com

Abstract

"Push" technology (also referred to as broadcast or multicast technology) is gaining rapid acceptance as one of the most important enabling technologies for large scale information delivery, transforming substantial portions of the commercial and financial activities in fundamental ways. The applications of this technology are numerous: financial products and services (algorithmic trading and real-time risk analysis), electronic commerce (document delivery over the Internet), customized news delivery and mobile computing to name a few. In the classical push based delivery model, a server broadcasts data objects on a shared downstream channel. Clients subscribe to objects, i.e., they pay for the right to download specific objects from the broadcast for specific periods of time. One of the most important notions in this context is that of *access legitimacy*: (a) a client should only be able to access items that it is subscribed to, (b) a client should only be able to access its objects of interest for its subscription period, and (c) the security system used must not be easily broken. We propose a system, SubScribe, that provides secure and efficient subscription delivery in this setting.

1 Introduction

Electronic commerce has received much attention lately [3]. One commodity that is particularly amenable to electronic commerce is *information.* In fact, such large quantities of data and information are already being "trafficked" over networks, that the term *information commerce* has been coined [5]. For example, consumers can subscribe to data servers that sell real-time financial information [9], weather information [25], documents and reports (such as the META group [20]) and a large variety of other information that buyers are interested in. There is a strong sense in the information commerce community that consumers' appetite for information is virtually boundless—what can be provided is only limited by available infrastructure (e.g., bandwidth availabilities, communication asymmetry [1]).

Given these infrastructural inadequacies, and the ever-increasing need to supply information, the traditional "pull" based technologies (i.e., where the data are actually queried and then received in response to this query) are increasingly deemed inefficient [2]. There are various reasons for this, including (a) the wastage of communication capacity by the very large number of clients asking for the same information, i.e., the same query being submitted repeatedly, (b) the amount of server capacity wasted in having to process these redundant requests and then (c) the same data objects being delivered repeatedly in response to these individual queries. One other factor that also makes pull schemes unattractive is the notion of communication asymmetry (see [2] for a discussion of various factors that contribute to this phenomenon), where upstream communication (i.e., client to server communication) is more expensive than downstream communication. Due to such limitations, "information-pushing" has been suggested as a key enabling technology for applications requiring large scale data dissemination. In a push-based environment, a server pushes (or broadcasts) information onto a shared channel (without explicitly waiting for client requests) and clients tune into these channels to download their items of interest. Recent advances in internetworking, wireless and mobile communications, and rapid enhancements of satellite, cable and telephone networks have enabled the deployment of this technology.

There has been research on push-based/broadcasting technology. Much work exists, both theoretical and practical, on broadcast networks from a communications perspective (see [30] for a nice survey). This work has mostly concentrated on developing the underlying communications technologies that make broadcast networking possible. This work is orthogonal to the content of this paper, and will not be discussed any further. There is also quite a bit of recent interest in studying broadcast methods from a data management perspective [22, 11, 40] (this is discussed in greater detail in section 2). These papers, which include prior work done by us, are mostly "framework" papers, i.e., they explore *how* to broadcast and *what* to broadcast. In fact, it would be fair to say that there has occurred a significant amount of learning on how to structure data broadcasts and we will apply this learning in this paper to construct the basic broadcasting infrastructure. However, several other problems remain to be solved to bring the concept of information commerce to fruition.

One such problem is providing *secure access control* for data items in broadcast schemes. To get a feel for this problem consider the classical broadcast environment, where an *information server* broadcasts to a large number of clients using a shared channel. Each broadcast consists of a number of data objects that clients are interested in. Each client is interested in a certain number of these objects and *subscribes* to them. Subscription refers to a contract that each client enters into with an agent, which entitles the client to access a data object for a specific period of time. Once the contracted period for a subscription is over, the subscription is considered to have *expired* and the client cannot access the data object any longer without resubscribing. Such a scenario is important in many applications. For example, financial data firms (e.g., Thomson Financial Services [16]) compute measures of interest called *analytics* (e.g., volatility and momentum). Such analytics reports are sold to a large number of consumers such as individual and institutional investors, financial sections of newspapers, brokerage houses etc. The number of reports produced (50-100) are quite small in comparison to the number of clients (more than 10,000) and each client typically subscribes to several of these reports. Clearly, such report distribution could benefit immensely from secure broadcasting technology. As another example consider a traffic report distribution service, where clients can tune in to a broadcast and download traffic parameters (e.g., congestion factors, average wait times at cross-streets) for various parts of a region (e.g., different sections of a city). If such a service were offered over an information commerce environment, people would most likely buy (i.e., subscribe to) the traffic parameters of the parts that they are concerned with (e.g., the parts on the route from work to home or the parts nearby their current location). Many such examples can be constructed for other applications, such as sports scores (people pay for those sports they are interested in), or news. The common feature of all these applications is that clients pay for, or subscribe, to certain items of interest from a broadcast of a large number of items. To enable the deployment of such applications, the following functionalities are necessary.

1. *A client must only be able to access the data items that it is subscribed to.* In other words, the access to all items that a client is not subscribed to must be blocked. An intuitively natural way to tackle this problem is by encrypting the data items.

2. *A client must only be able to access an item as long as its subscription to that item is not expired.* Clearly, in order access an encrypted item, a client needs to be provided with the decryption key. However, keys must be changed to avoid giving access to clients whose sub-

scriptions have expired. One obvious way of course is to change the security mechanism of a data item every subscription period. This is, however, prohibitively expensive, given the large number of *(client, subscribed_item)* pairs present in a system of reasonable size. Instead, we change keys for a data item only when a client has actually dropped out of their subscription for that item.

3. Finally, *the protocol(s) that implements the above two functionalities must provide an adequate level of security*, i.e., it should not be easy to breach the security provided by the access control mechanism.

Essentially, the problem is one of secure data management in broadcasts. While this problem has received virtually no attention from the database community, the underlying security problem is receiving much attention currently from the security community, following the publication of the now classic paper by Fiat and Naor [15]. Subsequent to the publication of this paper, a significant amount of work has appeared on broadcast security schemes [29] (more on this in section 2). However, much (if not all) of this work is concerned with one of two issues: (a) good encryption of broadcasts, or (b) broadcasting to static groups of users. Both of these, though related to the problem identified above, do not help solve them directly. In particular, we are interested in a practical system that uses existing cryptographic schemes to provide efficiency and security in a setting where subscription groups are dynamically changing, i.e. clients may join and leave subscription groups arbitrarily over time.

Given such an environment, in this paper we present a system, SubScribe, to help the simultaneous achievement of the three goals outlined above. Moreover, the protocols achieve this goal while imposing *very low* overhead on the server as well as the clients. As a result we tackle an important and hitherto unsolved problem in the domain of push based data dissemination and, hopefully, take a small step in bringing *information commerce* to fruition.

The rest of the paper is organized as follows: first, we discuss related work and analyze the strengths and weaknesses of existing solutions in section 2. We introduce some preliminary ideas used in our system SubScribe in section 3. We present the intuition behind the SubScribe system in section 4 and present details of SubScribe and a symmetric key variant in section 5. We present our simulation model for comparing different protocols in section 6 and discuss the results of the simulations in sections 7. Finally, we conclude in section 8.

2 Related Work

The research reported in the paper deals with secure data management in a broadcast context. Secure database systems and the management of data in such systems are an area of recent and significant interest. The primary goal of this research has been to enhance privacy in databases where, traditionally, simple security measures have been provided on a per-table basis. Primarily three types of information security mechanisms have been proposed: (a) *discretionary access control* which is of the kind of access control provided by current day file systems and databases [38], (b) *mandatory access controls* where every data object is assigned a security classification and every user is assigned a security clearance. Subsequently a user is granted or denied access based on the underlying authorization policy which dictates which clearance levels are authorized for which security classification. Such kinds of controls are prevalent in military applications. A strict form of mandatory access control is the *multi-level secure* (MLS) relational model [27, 14], and (c) *role based access control*, where the answer provided by the database depends upon who queries the data (e.g., a reporter may be provided a different answer than a nurse in answer to the following query submitted to a hospital database: `what is the diagnosis of person X`, where X is a famous person). A nice discussion of such type of security is provided in [38]. Recently there has been work on enforcing multiple access control policies in the same system [28]. In summary, there has been interesting work on secure data management. However, none of this work considers broadcasts, nor does this work consider subscription based services. Thus, this work is novel in those respects.

Another reference area of this paper is *broadcast data management*, which is of significant recent interest. Most of the work performed in this area has dealt with the following questions: (a) how does a server determine *what* to broadcast? and (b) how do clients efficiently retrieve data from a broadcast? A number of papers have appeared on this subject, including [22, 11, 1]. These papers include the mobile broadcasting work done at Rutgers, the *broadcast disk* work and some work that we have done recently on determining broadcast content. We borrow from the results of this work in the sense that we assume a basic broadcast structure that is in agreement with some of this earlier work. We augment this work by considering secure access control in this framework.

The topic of this paper is somewhat related to some of the issues in information security, such as encryption and cryptography. Traditionally, most of the papers in this area have proposed cryptosystems for point-to-point communication [19]. However, providing security for multipoint broadcasts is an issue of considerable current interest in the cryptology community. An important paper in this area is [15], which introduced new theoretical measures for assessing encryption schemes designed for broadcast transmissions. The authors also suggest schemes to broadcast a secret message to a privileged subset of any number of users out of a universe of n users such that no coalition of k users not in the privileged subset can learn the secret. The goal is to minimize the number of keys each user has to manage. It is shown that the most powerful of their protocols requires every user to store $O(k \log k \log n)$ keys. Our environment is considerably different. For instance, our user coalitions are mutually non-exclusive and dynamic, i.e., users may belong to more than one coalition and users join and leave coalitions arbitrarily (subscription). Moreover, the protocols that we come up with require users to store no additional keys (other than their own deciphering key, assuming a public key security system), i.e., the number of keys stored is a constant 1. Following the publication of [15] there has been a number of other papers on broadcast security. Some of these papers (e.g., [18]) deal with designing novel cryptosystems for broadcast communication, and are completely orthogonal to the topic of this paper. Yet others have attempted to come up with security schemes that allow secret messages to be sent to a privileged set of users [39, 26, 7, 6]. These papers are more related to our work, but the solutions offered, namely *group* and *session* keys, do not work well in our case (as presented in the next section), primarily because there is an explosion of the number of keys that have to be managed and distributed. Moreover, *none* of this work considers the subscription problem, which is a key problem in information commerce.

The notion of managing *transitory coalitions*, or *dynamic groups*, considered in this work, also comes up in the context of multicasting and fault tolerance literature. For instance, the idea of *quorums* is pervasive in studying fault tolerance in distributed systems [4]. A recent nice paper that combines ideas from multicasting and fault tolerant areas is [37]. In particular, that paper considers the ordering of multicast messages in a logical process ring network. However, the thrust of these papers is the issue of coordination—how to manage groups of users to ensure properties such as reliable message passing. Very little of this work, as far as we are aware, deals with secure data access. Moreover, the classical information pushing framework assumes a client-server architecture, where there is no communication among clients and very little communication from clients to the server. Most of the papers in multicasting and fault tolerant systems require large scale communication among the member set.

Finally, in the industrial world, there has been a flurry of activity for products that provide push technology in various forms. We note here products like TIB/ObjectBus from TIBCO [35], Velociti from Vitria [36] and SmartSockets from Talarian [34] all of which are publish-subscribe communication middleware. The model of multicasting is the one of channeling: the server receives data from publishers and broadcasts it to clients who subscribe to appropriate channels. Encryption occurs between a publisher and the server and between the server and subscriber which resembles the point to point encryption scheme described above. The emphasis of these systems is on transactional real-time messaging and message queueing for guaranteed delivery and support for sophisticated interactions between distributed applications. As far as we are aware, no existing products support the subscription based access control schemes studied in this paper.

2.1 Analysis of the Applicability of Multicasting Approaches To Our Problem

Three basic approaches to multicasting may be identified from the literature: (a) point to point communication, (b) using *session keys*, and (c) using *group keys*. We now discuss the advantages and disadvantages of these approaches in our scenario. We assume for this discussion that a public key cryptosystem. In a public key system, each client has two keys, a *public key* (also known as the encryption key) and a corresponding *private key* (also referred to as the decryption key). The server encrypts items with clients' public keys while clients decrypt items with the corresponding private keys.

The *point to point* approach to broadcasting mandates that the server must perform encryption of each object that each user is subscribed to and send the ciphertext separately [19]. For example, if data item D_i is subscribed to by c clients, then the server must include in the broadcast c encryptions of D_i, one meant for each of the c clients. Thus for a system with N users $U_1, U_2, \ldots, U_N$, where user U_i is subscribed to n_i data objects in the current broadcast period, the server must include $\sum_{i=1}^{N} n_i$ ciphertexts in the broadcast. This is, clearly, very inefficient as multiple ciphertexts corresponding to the same data item must be sent.

In the *session keys* scheme [8], each broadcast is considered a session and in each session, i.e., in every broadcast, a *session key* is assigned to each subscriber group. In other words, there is a session key corresponding to each data item. These keys are arbitrarily generated for a specific session, and then discarded. A new set of session keys is generated for each broadcast. The encryption key for data item i is denoted with $SessEncKey_i$, and the decryption key for that data item is denoted with $SessDecKey_i$. Under this scheme, the server enciphers each data item that must be broadcast, say $D_1, D_2, \ldots D_m$, with their corresponding encryption keys, i.e., with $SessEncKey_1$, $SessEncKey_2$, $\ldots$, $SessEncKey_m$, respectively. However, now the server needs to send the decryption keys to the appropriate subscribers, i.e., to each member of SS_i the server needs to send $SessDecKey_i$. To do this the server constructs the ciphertext of each decryption key with the private key of each individual client. In other words, if SS_i consists of clients $C_1, C_2, \ldots, C_k$, and we denote the public key of C_j by $PubK_j$, then the server computes ciphertexts $R_1, R_2, \ldots, R_m$ where $R_j = PubK_j(SessDecKey_i)$. This is done for every member of each subscriber set and the server concatenates these ciphertexts together. One advantage of this method is that the key management overhead is minimal as the session keys are thrown away and new keys are generated for the next broadcast. In addition, this approach naturally solves the subscription expiry problem as session keys are changed each broadcast. However, this approach has the problem that the session key must be communicated to all subscribers in every broadcast.

A third approach to secure broadcasting is to use *group keys* [24]. This technique may be applied to our scenario as follows: let the subscriber set of (i.e., the set of all clients currently subscribed to) data item D_i be denoted by SS_i. Clearly, there exists, for each data item, a subscriber set (some of which may be empty). Now assign a group key to each subscriber set. We use a symmetric cryptosystem (e.g. DES) for the group key as this is more suitable for encryption of large data items than public key cryptography. The data key for a group is made known to each member of the group. Under this model, the server can broadcast each item enciphered with the current data key, and each member of the group deciphers the item with the same data key. Although this method requires more key management than either the point-to-point approach or the session key approach, it has advantages over both of them. First, this method alleviates the *multiple ciphertext* problem of the point-to-point approach. Second, the group key will only be changed when a client drops out of a subscription, so new keys only need to be communicated once to each client, rather than in every broadcast.

From the above analysis one can identify a set of desirable properties of "good" access control protocols for secure broadcasting.

1. *A good protocol should not impose a large key management overhead*, i.e., neither the clients nor the server should be burdened with managing a lot of keys. Ideally, the server should just deal with the public keys of clients, and the client should just have to remember their own private keys.

2. *A good protocol should not impose a large processing overhead upon the client* (as in the session key method outlined above). In the ideal case each client should have to do just one deciphering, i.e., simply decrypt the data item. Note that a really good security protocol would cause this deciphering to be low-cost. However, this falls in the domain of encryption schemes which is not a concern of this paper.

3. *A good protocol should not impose a large space overhead on the broadcast.* This is especially important in broadcasting in energy restricted data access environments such as mobile computing environments. In the above discussion, the point to point method violates this goal in an extreme fashion. The session key method also causes fairly significant space overhead.

4. *A good protocol should provide a low cost solution to the subscription expiry problem*

As the reader will see, we design methods that most closely resembles group keys, with some of the benefits of a session key approach.

3 Preliminaries

We first describe the basic context which is assumed by the ensuing discussion. A server broadcasts data items (which are seen as arbitrary stream of bits) over a shared communication channel. The channel is not restricted to a particular media or network type; for instance, it could be air, LAN, or the Internet. We assume that each data item is identifiable with some sort of identifier.

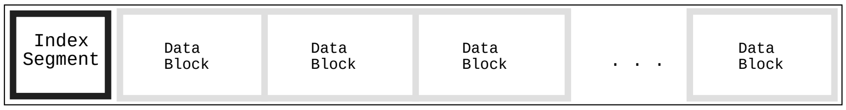

Figure 1: Broadcast structure

Figure 1 shows the broadcast structure we assume in this paper. It is is composed of a number of indexed data blocks. We choose a simple indexing strategy—the index is broadcast upfront in the broadcast. Note that there have been several indexing strategies proposed for efficient access from broadcasts (see [22] for a nice treatment of indexing in broadcasts)—the specifics of the indexing strategy has no impact on our security protocol. The data blocks correspond to specific data items. In secure broadcasting schemes, the data item will be enciphered in some way. Clients tune in to the top of each broadcast and get pointers to their subscribed items. Then they tune out and tune back in again to download. The protocols that we develop will add a security layer on top of the basic broadcasting model described above. This access control layer will involve encrypting the data items and then adding smart controls on top of the encryption logic to achieve the four goals enumerated above.

Clients subscribe to specific data items, i.e., they are only allowed to access data items they are subscribed to, for the duration of the subscriptions. Note that a specific subscription is applicable to a *(Client, Data Item)* pair. The subscription process is not of concern in this paper and is assumed to be carried out autonomously. *Autonomous agents* are a much investigated current research area [21]—these appear to be particularly suitable to execute the subscription mechanism. For our purposes, a new subscription causes the server to become aware of a client's right to access a particular data item for a specific period of time. Finally, we note that several classes of information that are likely candidates for broadcasting are dynamic, e.g., financial analytics, weather, traffic. In such cases the client will usually wish to be appraised of the most recent value of its data items of interest. As a result, they will in many cases, want to download their data items from every broadcast.

Our goal is to design information delivery and access strategies in this environment. Our communication model implements public key and symmetric key encryption [31] as a means to enable secure communication.

4 Basics of SubScribe

Our basic approach in designing SubScribe is to exploit the best features of the group key approach and the session key approach. A table of notation used in the description of SubScribe is listed in Table 2. SubScribe uses two types of keys: (a) *client keys*, and (b) *data keys*. For the client keys, we use a public key cryptosystem such as RSA [32], since clients may already possess public/private key pairs for use in any secure application, and a public key certificate directory could be used to look up these keys. Of course, if such keys do not exist, they can be generated as part of an initial registration period with the subscription service. For data keys, we use a symmetric system such as DES [33], since this is more suitable for bulk data encryption. The public key of client C_i, which is assumed to be known by the server, is denoted by PK_i. C_i's corresponding private key, which assumed to be secret to C_i, is denoted by SK_i. Messages encrypted with PK_i can only be decrypted by C_i using SK_i. Each data item D_i has a data key, denoted by DK_i, for use in the symmetric system. The data keys are initially known by the server only, but will also be securely transmitted to subscribers of D_i. New data keys will be chosen as needed to ensure that only current subscribers can read a particular broadcast.

More specifically, when broadcasting D_i, the server encrypts it with DK_i, producing the ciphertext $DK_i(D_i)$ of D_i, denoted by T_i. T_i is included in the *data component* of the data block corresponding to D_i. Subsequently, only clients knowing DK_i are able to access D_i. Thus, the data key DK_i is included (in an encrypted fashion) in the *key component* of the data block corresponding to D_i. In order to provide maximum efficiency, the data key is only changed when clients drop their subscriptions—this will become clearer when we describe the details of SubScribe in the following section.

The detailed broadcast structure that is required by SubScribe is shown in Figure 3. As discussed before, the broadcast starts off with a *broadcast index*, followed by a sequence of *data blocks*. The broadcast index segment and all the data blocks contain an *orientation header* (OH). The OH consists of a single element, namely an offset to the start of the next broadcast. This pointer is intended as a "tuning-aid" for clients (recall our earlier comment about the dynamism of the underlying database—clients are typically interested in downloading from all broadcasts during the course of their subscriptions). Thus, when a client is finished downloading from the current broadcast, it can sleep until the next broadcast. The OH enables a client to know when the next broadcast is scheduled to commence. Moreover, this pointer also helps clients during their initial probe into the broadcast (i.e., the first probe subsequent to registering with the SubScribe system).

Notation	Meaning
C_i	client i
PK_i	public key of C_i
SK_i	private ("secret") key of C_i
D_i	data item i
DK_i	data key of D_i
SS_i	subscriber set of D_i
DK_i^p	DK_i for broadcast period p
T_i	ciphertext of D_i encrypted with DK_i, i.e., $DK_i(D_i)$
$R_{i,j}$	a data key for D_j encrypted with C_i's public key, i.e., $PK_i(DK_j)$
SS_i^p	SS_i for broadcast period p
OH	Orientation Header
BI	Broadcast Index
DBI	Data Block Index
OKS	Offset to Key Segment
OTI	Offset to Item
KDK	Key Distribution Key
cid	Client id

Figure 2: SubScribe notation

Essentially, if the client misses the initial index, it reads the next OH, and tunes out until the start of the next broadcast, when it can read the index.

As mentioned before, the *broadcast index* (BI) precedes everything else in a broadcast. The BI consists of index records that hold pointers to each item's data block in the current broadcast. More specifically, an index record consists of a 2-tuple $\langle$*item id, offset to data block*$\rangle$. A client obtains pointers to its desired data items from the BI, and then sleeps, only to wake up at the desired points in time. We remark that one can easily come up with much more efficient schemes (e.g., repeating the index in a broadcast). However, our intent in this paper is not to examine ways of indexing broadcasts, so we adopt this scheme for simplicity.

The BI is followed by a sequence of data blocks. Essentially, a data block contains all information required by a subscribed client to download a specific data item—it contains an encrypted data item, and related key information necessary to decrypt this item. More specifically, a data block consists of three parts.

1. First, it has a *data block index* (DBI) that holds pointers to the key information and to the data item itself. Basically, a data block index record contains the 3-tuple $\langle$*client id, offset to key segment (OKS), offset to item (OTI)*$\rangle$. The OKS element associated with client id C_i indicates where C_i can find the key information (i.e., the data key as described before) that

will enable it to decrypt the data item in this data block (this presupposes that C_i is subscribed to this item). As will be clear from the next section, a data block will contain key information for the various clients subscribed to this item. The OKS element will aid clients in efficiently getting the key information chunk specifically meant for them. The OTI element simply points the client to the top of the encrypted data item in the current data block. The idea is that a client, say C_i, will read the DBI tuple meant for itself, and will know where it needs to tune in to download the appropriate key and where to tune in to download the encrypted data item.

2. The second part in a data block is the *key component*. Essentially, the key component contains key chunks meant for the various clients subscribed to the data item in the current data block. In other words, if n clients are subscribed to this data item, the key component will consist of the concatenation of n key chunks. The OKS element described in the previous item points to these key chunks. The precise content of the key component will be made explicit in the next section.

3. The last part of a data block is the *data component*. This contains a data item encrypted with its data key.

5 Details of Subscribe

We are now in a position to describe SubScribe in procedural fashion. The full broadcast structure is as shown in Figure 3. Note that the figure is not drawn to scale and reflects the order of broadcast components rather then their sizes. The protocol has two components: a server side or *information delivery* component and a client side or *information access* component. These are shown in Algorithms 1 and 2, respectively.

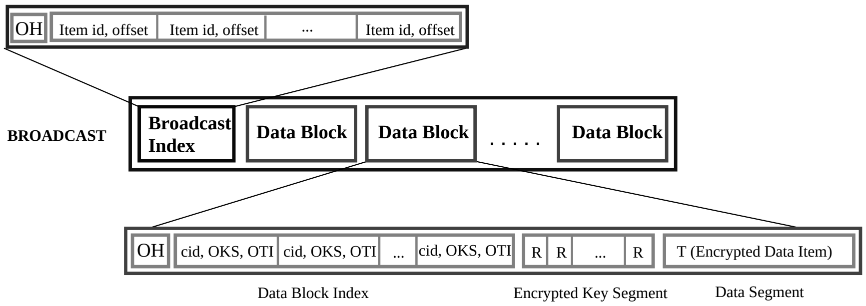

Figure 3: Detailed view of broadcast structure

The server side protocol is responsible for the delivery strategy for data items. It distinguishes between two types of data items: (a) data items whose subscriber set in the current broadcast includes every client who were subscribed to this item in the previous broadcast as well—we will refer to these items as NODROP items, and (b) data items not satisfying the previous criterion, i.e., items which have lost some subscribers in the current broadcast. We will refer to these as DROP items. To include a DROP item, say D_i, in the broadcast, the server chooses a new data key, DK_i, and creates a data block for this item as follows:

- **Data Component:** The server encrypts D_i with DK_i and includes the ciphertext in the data component
- **Key Component:** For each client in the subscriber set of D_i, the server encrypts DK_i with the client's public key and includes the ciphertext in the key component. In other words the key component of DROP items essentially becomes a concatenation of ciphertext chunks, where each ciphertext chunk represents an encryption of DK_i with a specific subscriber's public key.

To include a NODROP item into the broadcast, the server uses the same data key that was used in the previous broadcast for this data item. This is possible due to the fact that using this data key does not compromise security—all prior subscribers are still subscribed to this item. Also, in this case (potentially) substantial savings are realized as the key need not be sent to all the prior subscribers. The server composes the key component by encrypting the data key *only for the new subscribers*. Existing clients are notified of the fact that the data key remained the same by inserting a special offset value of -1 in the *data block index* record for that client. In both the DROP and NODROP cases, index records are created by the server for constructing the two types of indexes described before. This creation process is shown in our algorithm.

One clarification needs to be made regarding the way SubScribe treats NODROP items. According to the server side protocol just described, if no subscriber drops out, the key remains the same. In practice, data keys should always have a limited lifetime, and a new data key should be chosen for a particular data item if the key has been used for the full lifetime, even if no subscribers have dropped out. Actual key lifetimes should be chosen to depend on the perceived value of the data to would-be attackers.

The client side extracts the relevant information from the broadcast as shown in Algorithm 2.

5.1 A Symmetric Key Variant

The SubScribe mechanism, as described in the previous section, encrypts data keys with client public keys. In a dynamic subscription environment, i.e., where subscriber sets change substantially from broadcast to broadcast, it is possible that the size of key components will turn out to be significant.

Algorithm 1 The SubScribe Information Delivery Protocol: current broadcast period is p.

Input: $\mathcal{D}$: a set of data items for inclusion in the broadcast
$\mathcal{D}_+$: $\{D_i \in \mathcal{D} \ : \ SS_i^p \supseteq SS_i^{p-1}\}$ `/* NODROP items */`
$\mathcal{D}_-$: $\mathcal{D} \setminus \mathcal{D}_+$ `/* DROP items */`
For each client C_i in the system, the client's public key PK_i

Do:
  for all $D_j \in \mathcal{D}_-$ **do**
    choose random DK_j^p `/* new key required */`
    generate T_j, the ciphertext of D_j, $T_j \leftarrow DK_j^p(D_j)$
    for all $C_i \in SS_j^p$ **do**
      $R_i \leftarrow PK_i(DK_j)$
      Add (i, R_i) to data block index. `/* Offsets will be added later. */`
    end for
    Create data block header using special *all* symbol.
    Add j and appropriate offset to broadcast index segment.
  end for
  for all $D_j \in \mathcal{D}_+$ **do**
    $DK_j^p \leftarrow DK_j^{p-1}$ `/* reuse previous key */`
    generate T_j, the ciphertext of D_j, $T_j \leftarrow DK_j^p(D_j)$
    for all $C_i \in SS_j^p \setminus SS_j^{p-1}$ **do**
      $R_i \leftarrow PK_i(DK_j)$
      Add (i, R_i) to data block index. `/* Offsets will be added later. */`
      Create data block header with offset to data.
    end for
    Create data block with orientation header, data block index segment with appropriate offsets added, key segment, and encrypted data item T_j `/* Set offset to key segment for previous clients to` -1 `*/`
    Add j and appropriate offset to broadcast index segment
  end for
  Create broadcast by concatenating orientation header and broadcast index segment with data blocks

Algorithm 2 Information Access Protocol: executed by each client, C_i. Current period is p.

Do:
  for all subscribed data item D_j **do**
    Download data block for D_j
    if C_i was not subscribed to D_j in period $p-1$ **or** data header contains *all* symbol **then**
      $DK_j \leftarrow SK_i(R_i)$ `/* Decrypt` R_i `to get data key */`
    else
      $DK_j^p \leftarrow DK_j^{p-1}$ `/*` C_i `already knows data key */`
    end if
    $D_j \leftarrow DK_j(T_j)$ `/* Decrypt data item */`
  end for

This motivated us to look for ways to reduce this overhead. This lead us to the development of the *symmetric key* variant of the SubScribe protocol, that we will refer to as SubScribe^S. In SubScribe^S, the server gives each client a symmetric key (e.g. a DES key) to be used for communicating new keys.

In this approach, clients would be given a *key distribution key* (KDK) by the server when they first join the service (e.g., in the header of their first received broadcast). Subsequently, this key distribution key would be used in place of the public and private key in the approach detailed in Section 5. In other words, unlike the SubScribe protocol, where data keys are encrypted with client public keys, in SubScribe^S, the data keys will be encrypted with client KDKs. There are two advantages to using such a symmetric key. First, the KDK is smaller than the keys in the public key cryptosystem. Second, the symmetric encryption and decryption operations are faster than the corresponding public key cryptography operations. In many public key cryptosystems 1024 bit keys are used. Encrypting data keys (which would typically be of the order of 64 bits) using these public keys will result in 1024-bit ciphertexts. However, if we use the 64-bit KDKs to encrypt the data keys, then the encrypted 64-bit data keys will also 64 bits—almost an order of magnitude less than public key encryptions. Admittedly, such KDKs require an additional key exchange step. However, this step only needs to be done once for a new client and can be combined with a broadcast, as illustrated by the following example.

Consider a new client C_i who is interested in items D_j and D_k, where D_j appears before D_k in the broadcast. Then the key component of the data block for D_j will include a ciphertext chunk meant for C_i. Usually, this ciphertext will be the data key of D_j enciphered with the KDK of C_i. In the first broadcast only, the ciphertext will be the encryption of both the KDK for C_i and DK_j, enciphered with the public key of C_i. Note that since the symmetric KDK and data keys are much smaller than the public key, (e.g., size(KDK) + size(data key) $\leq$ size(public key)), it is possible to concatenate the two keys and encrypt them as a single block. *In other words, the KDK distribution, for all intents and purposes, comes for free.* Thus, we can guarantee that given identical content, the size of the broadcast in the SubScribe^S protocol, will be always less than or equal to the size of a pure SubScribe broadcast.

6 Simulation Model

To empirically verify the efficacy of our protocols, we ran detailed simulations of SubScribe and SubScribe^S in a wireless broadcast environment. Note that we could have assumed other networking infrastructures as well, such as a broadcast LAN or the Internet. However, our choice of a wireless network was motivated by two reasons: (a) a large number of applications of our proposed technology fit naturally in the wireless environment (e.g., traffic and financial reports to mobile users), and (b) we already have experience in the wireless domain [13]. It is important though, to evaluate the performance of our system under different environments.

Our simulation programs were written in SIMPACK, a C/C^{++} based simulation toolkit [17]. Our simulation model contains three major components: (a) a database component, (b) A client component, and (c) a server component.

6.1 Database and Broadcast Model

The database in our simulations consists of d data items. Each data item is of a specific size. The size of a data item is generated from a uniform distribution within the range $[MinBytesPerItem, MaxBytesPerItem]$. Based on the above model, it may be seen that our data items have very general semantics—they may be composed of records, or may be unstructured, such as a document.

Each broadcast is composed of all items that are currently subscribed to by clients. Inside a broadcast, items are ordered by popularity, i.e., the number of clients subscribed to a given item. Thus, before any given broadcast, the server knows exactly which items are to be included in this broadcast, and where each item is located in the broadcast. A corollary of this is that the server knows the broadcast size, which enables it to indicate the time of onset of the next broadcast in the current broadcast. If a client's subscription to a data item is scheduled to expire in the middle of a broadcast before downloading the data item, then the client will not be allowed to access this item. The broadcast structure we model in our experiments is exactly the structure shown earlier in Figure 3. The two primary components in this structure are index and data. Moreover there are two types of index as shown earlier: a *broadcast* index (that indexes data blocks in the broadcast) and a *data block* index, contained inside (and at the top of) each data block, which holds pointers to data item keys and data items as described earlier. The only decision to be made in modeling these indices is to determine the sizes of index records. We model each broadcast index record to be *BcastIndexBytes* bytes and each block index record to be *BlockIndexBytes* bytes. Note that, for the *Point to Point* protocol, we only need a *Broadcast Index*. However, each index record must consist of a 3-tuple, $\langle$*item id, client id, offset to item*$\rangle$. The size of the *Point to Point* index record is denoted with *PTP-IndexBytes*. We model the index itself in linear record fashion—more sophisticated index structures such as B+ trees are possible but we do not consider them in this paper.

In addition to the indices, we also need to model the size of encrypted data items and encrypted keys. In order to encrypt with either RSA and DES with a k-byte key, a b-byte item must first be broken into blocks of size at most k. Since both RSA and DES are length-preserving, the resulting ciphertext is modeled as $\lceil \frac{B}{k} \rceil \times k$ bytes.

Our database model assumes locality of reference. A fraction of items, defined by the parameter *PercentHot*, are designated as *hot* items. These items will be more heavily requested by clients than *cold* items. Clients will request a hot item with probability *ProbHot*.

In addition to the SubScribe and SubScribe^S protocols, we also simulate the point-to-point and session key protocols. These models are created according to the description of these mechanisms provided earlier in the paper.

An important point to note is that in our model the broadcast size is variable (i.e., broadcasts are not periodic). This occurs due to two reasons (a) varying content in different broadcasts, and (b) different sizes for different items. Also note that given the same subscription profile, the different delivery mechanisms will yield different broadcast sizes, as key management is handled differently in these mechanisms. In fact, it is precisely this difference that we wish to study in our experiments.

The database is updated asynchronously, and, clients are interested in downloading the entire data item at each broadcast within its subscription period. A download is said to be complete when the data item is fully read by the client.

6.2 Client Model

In our model a client has a set of specific data items of interest. *NumClients* denotes the number of clients in the system. We set this variable to different values for different experiments to observe the system under different loads. We generate all the clients at the start of each simulation run along with their subscription profiles. For each client, we first generate the number of data items the client is interested in. For each client this is generated from a Uniform distribution with parameters [*MinItems*, *MaxItems*]. After generating the number of items client is interested in, we generate the individual data item and its subscription information by generating a 2-tuple consisting of the following:

- An *id*. This is generated by drawing randomly, without replacement, from the set of item ids available in the database.
- An initial subscription period. This is generated from a truncated normal distribution with parameters [*MinSubsPeriod*, *MeanSubsPeriod*, *MaxSubsPeriod*, *StdSubsPeriod*].

The initial subscription starts after a random wait period at the beginning of the simulation.The server constructs the initial broadcast according to the initial subscription profiles constructed as above. Subsequently, as soon as the subscription of a specific client to a specific data item expires, the client waits some amount of time, and resubscribes to the object with a new subscription period generated as above. The random wait period is generated from an exponential distribution with the parameter *MeanWaitPeriod*.

Further, each client owns a Public Key and Private Key pair of *PublicKeyBits*. For the SubScribe^S protocol, the client is given a much smaller key, of size *KeyDistributionKeyBits*, additionally.

After downloading the encrypted data key and the encrypted data item, the client must perform decryption to recover the key and the data item, respectively. Decryption involves a cost incurred by the CPU of the user's computer. However, we do not model the cost of decryption for the following two reasons: a) The data item will have to be decrypted in all the protocols, thus the same cost will be incurred for each protocol, b) The time for decrypting a key chunk is in the order of milliseconds, thus negligible. In fact, with appropriate design, the decryption process could occur simultaneously with the downloading process, essentially incurring no appreciable additional time cost at all.

Data access by clients from a broadcast is modeled according to the client side protocols described previously in the paper.

6.3 Energy Model

In a wireless computing environment, especially when clients have to access data without the benefit of a constant power source (such as in mobile environments), a major constraint is the energy consumption of the computing unit. Clients have to recharge their batteries frequently, and do not want to consume energy unnecessarily. A good data access protocol should keep energy expenditure low. Therefore, in our simulations, we are interested in the energy consumed by each protocol. In the following we describe the energy model used in the simulations.

A computing unit (e.g. a palmtop or laptop) used by the client may be equipped with four major energy consuming components: the **disk drive**, the **CPU**, the **display unit**, and the **mobile data card** (MDC). In our model, clients are *diskless*. In the diskless state, clients do not write the retrieved data to disk; rather, they cache it in memory. This modeling assumption is motivated by the fact that we consider data items whose states are dynamic, clients are interested in the most recent value of the item, and want to download from each broadcast. Therefore, energy expenditure can result from three sources: the CPU, the display unit and the mobile data card (MDC). We make three additional assumptions about the client units:

1. At the start of the system, all clients are powered up, and they remain powered up throughout the simulation.

2. Other components of the mobile unit, e.g., random access memory, consume power at a constant rate, and are therefore of little interest.

3. All uplink communications are handled by an extraneous service, therefore the energy expenditure for transmissions are not considered in the model.

Next, we describe each of the power consuming components that we are considering. The energy expenditure is calculated using the equation ENERGY (Joules) = POWER (Watts) $\times$ TIME (seconds). Table 1 shows typical energy consumption rates for these components and states. These numbers are taken from real product specifications [23, 10].

Component	**State**	**Rate of Consumption**
CPU	Active	0.25 Watts
CPU	Sleep	0.00005 Watts
Display	Active	2.5 Watts
Display	Off	0 Watts
MDC	Receive	0.2 Watts
MDC	Sleep	0.1 Watts

Table 1: Component States and Power Consumption

CPU The mobile unit's CPU can exist in one of two states: *active* (A), and *sleep* (S). The CPU is in the A state when any operation is taking place in the mobile unit, i.e., any other component is in the *active* state. Otherwise, the CPU is in the S state.

Display The display component can exist in one of two states: *on* (N), and *off* (F). In the N state, the display unit is actively displaying data on the screen. In F state, the unit displays no data, and consumes no power. We assume that the display is in the F state at the start of the simulation. The display enters the N state while the mobile unit is downloading the data items from the broadcast, and returns to the F state at the completion of data download.

Mobile Data Card The mobile data card (MDC) has two states: *receive* (R), and *sleep* (S). In R state, the unit is reading from the broadcast, and in S state, the unit is neither transmitting nor receiving. Initially, we assume the MDC to be in the R state. It enters the R state for index reads and data reads, and returns to the S state immediately upon completion of the read.

We assume that each mobile unit is equipped with a wireless modem for accessing bandwidth on air. We assume a data rate *TransRate* of 19.2 Kbps, the rate at which clients can receive data from the broadcast.

6.4 Server Model

To model the server, we implement the broadcast protocols described earlier in detail.

6.5 Performance Metrics

The following are the three performance metrics that we use to evaluate the protocols.

- Average Broadcast Length (ABL): A critical performance metric is *access time*, which measures how long clients have to wait to access their items of interest. A meaningful measurement of access time, however, is complicated in our case for two reasons: (a) clients access data repeatedly from successive broadcasts, (b) clients subscribe to different items. Therefore, we set out to design a metric that captures how rapidly clients can access data items. In our model, every subscribed item is included in the broadcast. Thus, if a client is subscribed to an item, it is guaranteed to find it in the current broadcast. Another way of saying this is to state that clients access data items once every broadcast during the time they are subscribed to these items. Thus, the longer the broadcast, the longer a client will have to wait before receiving its data item again. A natural metric thus, is the *average broadcast length* (ABL), which is nothing but the broadcast length averaged over the length of the simulation.

- Normalized Energy Expenditure (NEE): NEE measures the energy spent by the clients while using each protocol. NEE is simply the energy spent on average by a client to download a data item. For a client C_i, if the energy spent throughout the entire simulation is W_i, and if he accessed his data items of interest n_i times, then his normalized energy expenditure NEE_i would be $\frac{W_i}{n_i}$. Therefore, for a total of $NumClients$ clients, $NEE = \frac{\sum_{i=1}^{NumClients} W_i}{\sum_{i=1}^{NumClients} n_i}$

- Key Management Overhead (KMO): Another dimension along which protocols differ is the number of keys the clients and the server have to manage. Therefore, we include a comparative discussion of the protocols in terms of the overhead caused by key management.

7 Performance

We first report the results of baseline experiments and then study the sensitivity of these results towards different parameters.

Notation	Meaning
d	Database Size
PercentHot	Percentage of Hot items
ProbHot	Probability of selecting a hot item
NumClients	Number of Clients
MinItems	Min number of items per client
MaxItems	Max number of items per client
MinBytesPerItem	Min size of item
MaxBytesPerItem	Max size of item
MeanSubsPeriod	Mean subscription period
MinSubsPeriod	Minimum subscription period
StdSubsPeriod	Std. dev of subscription period
MeanWaitPeriod	Mean time between successive subscriptions
BcastIndexBytes	Index Record size
OrientationHeaderBytes	Orientation Header size
BlockIndexBytes	Data block index record size
PTP-IndexBytes	Index Record size for Point to Point protocol
PublicKeyBits	Number of bits in a Public Key
KeyDistributionKeyBits	Key Distribution Key size
ABL	Average Broadcast Length
NEE	Normalized Energy Expenditure
KMO	Key Management Overhead
TransRate	Transmission Rate

Table 2: Notation for the Simulation Model

7.1 Baseline Experiments

We simulated our model over a period of 8 hours by varying the number of clients (i.e. the load) in the systems. The first three hours are "thrown away" to ensure the system enters a steady state (appropriate statistical significance tests were conducted). For each set of parameters, we produced average broadcast length (ABL) and normalized energy expenditure (NEE) curves, and plotted the performance of each protocol on the same figure. Note that the axes in all ABL figures are the same in order to facilitate comparison across experiments. The same is true for the NEE figures. The results of the baseline experiments are shown in Figure 4.

7.1.1 Discussion of the Average Broadcast Length Curves

Let us first consider the Average Broadcast length (ABL) curves shown in Figure 4A. The point-to-point (PTP) protocol shows a linear increase in

Parameter	**Value**
d	3000
$PercentHot$	20
$ProbHot$	80
$NumClients$	varies between 100 and 1000
$MinItems$	3
$MaxItems$	6
$MinBytesPerItem$	1K
$MaxBytesPerItem$	5K
$MeanSubsPeriod$	45 min
$MinSubsPeriod$	1 min
$StdSubsPeriod$	0.5 min
$MeanWaitPeriod$	90 min
$BcastIndexBytes$	3
$OrientationHeaderBytes$	1
$BlockIndexBytes$	4
$PTP\text{-}IndexBytes$	5
$PublicKeyBits$	1024 bits
$KeyDistributionKeyBits$	64 bits
$TransRate$	19.2 Kbps

Table 3: Baseline Parameters for the Simulation

ABL with increasing client load. This is expected, as each new client results in the addition of as many data blocks to the broadcast as the number of items this client is subscribed to. Thus, holding the average number of subscription fairly constant, it is easily seen that the broadcast length is linearly proportional to the number of clients as shown in the figure.

More interesting are the ABL curves for Session Key (SK), SubScribe and SubScribe^S. We first discuss of the general shape and then comment upon the relative positions of these curves. All three protocols yield increasing curves with a similar concave shape, i.e., the slope increases more rapidly at lower loads than at higher loads. The reason for this lies in how the broadcast content changes at different loads. When the system has few clients, the broadcast typically contains a (relatively) small number of data blocks. At these low loads, the addition of new clients usually results in the addition of new data blocks to the broadcast. This happens because at low loads, new clients ask for data items which are not in the broadcast. In other words, at low client loads, the introduction of new clients usually results in significant increases in broadcast size. This explains the relatively high slope of the ABL curves at low loads. As the load gets higher

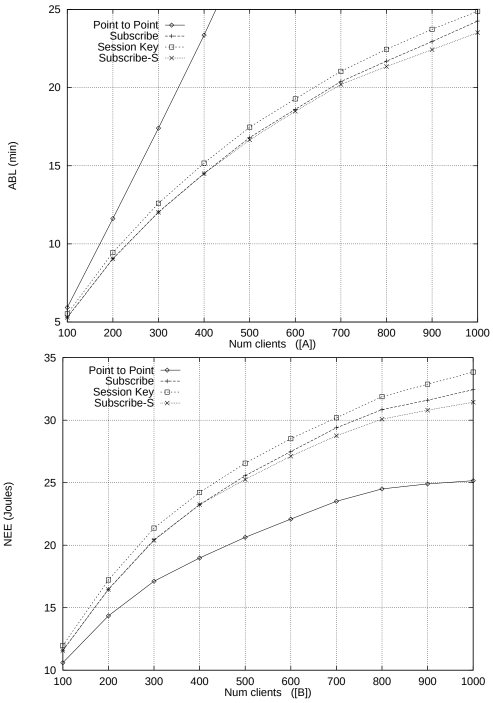

Figure 4: Baseline Results: [A] ABL for various client loads; [B] NEE for various client loads

however, the broadcast gets bigger and bigger, signifying that new clients end up asking for items that have a good chance of *already being in the broadcast*. Thus at higher loads, the introduction of new clients does not typically result in the addition of new data blocks in the broadcast; rather, a new client request will usually result in the addition of an index record to the data block indexes of each of the items requested by this client. This explains the much slower increase in broadcast sizes at higher loads.

Now we turn our attention to analyzing the relative performance of the ABL curves corresponding to the SK, SubScribe and SubScribe^S. Both SubScribe and SubScribe^S yield better broadcast sizes than SK. This is expected as the SK protocol usually generates larger key components for each data block. Recall that our protocols (both SubScribe and SubScribe^S) distinguish between DROP and NODROP items. This distinction results in significant decreases in the key component size—the key is only encrypted for the new subscribers for NODROP items. Actually, the effect here is somewhat muted as average subscription length has been chosen to be not very much larger than average broadcast size. For example at midrange loads, say *NumClients* = 500, the ABL ($\approx$ 18-20 minutes) is a little less than half of the mean subscription time (45 minutes) for these experiments. This means that the overall rate of unsubscribing is non-trivial, and, consequently, the proportion of DROP items are high. As we show in a later set of experiments, when subscription periods are high, SK tends to perform much worse than our protocols.

Finally, we remark on the difference between the SubScribe and the SubScribe^S curves. Essentially, the two protocols yield identical broadcast sizes until midrange loads (*NumClients* $\approx$ 500), after which the curves diverge—SubScribe^S yields better broadcast sizes than SubScribe. To see why this is so, consider the composition of each data block. For all practical purposes, the two major parts inside a data block are the *key component* and the *data component*. Clearly, for any data item included in the broadcast, the sizes of the data component will be the same for both SubScribe and SubScribe^S. The difference between the two protocols lies in the size of the key component. Effectively, SubScribe^S has a greater likelihood of producing a smaller key component as it uses a smaller key. This does, indeed, turn out to be the case. However, at low client loads the key component is fairly small (as, on average, few clients are subscribed to each item). Consequently the data component dominates the size of the data block. And, as mentioned before the sizes of the data block is the same for both protocols. At high loads however, the number of clients subscribed to each item increases significantly. This results in (comparatively) large key components. In these cases SubScribe^S derives advantages as each key chunk is expected to be smaller than the size of the corresponding key chunk produced by SubScribe. This results in increased separation of the curves with increasing load. To verify this claim, we measured the fraction of the data block occupied by the key component. This measurement yielded results that back up the claims made above. Due to space constraints of this paper we do not provide these histograms (see [12] for these results).

7.1.2 Discussion of the Normalized Energy Expenditure Curves

The NEE values portray how efficiently the different protocols manage energy consumption at clients. This is important as wireless and/or mobile clients are expected to be an important class of consumers for information commerce technology. Having examined the ABL results, the NEE results are easier to explain.

The most remarkable aspect of the curves shown in Figure 4B is the complete reversal of performance order with respect to PTP. PTP had the worst ABL results, but have, by far the best NEE values. This is easily understood when one considers the fact that PTP provides for the *smallest data blocks* of all the protocols. This is so as any data block in PTP only consists of the encrypted data item, while all the other protocols need to include a data block index and key information as well. Thus, clients spend less energy as they have to be tuned in for lesser amounts of time. In fact, as mentioned earlier, the key component becomes substantial at high loads. Thus, one would expect a dramatic difference between the PTP and the other curves at the highest loads. However, as is apparent from the figure, that is not the case—the separations remain modest even at high loads. To understand the reason for this, consider the fact that when the client is not actively tuned in to the broadcast it is in the *sleep* mode. However, as shown in table 1, in section 6, even in the sleep state clients consume energy. Moreover, the size of the PTP broadcast dramatically increases at high loads. Thus, clients end up consuming large amounts of energy while sleeping. This offsets, to a large degree, the savings accrued by downloading small data blocks.

The other curves exhibit predictable performance.

7.2 Effect of Varying Database Size

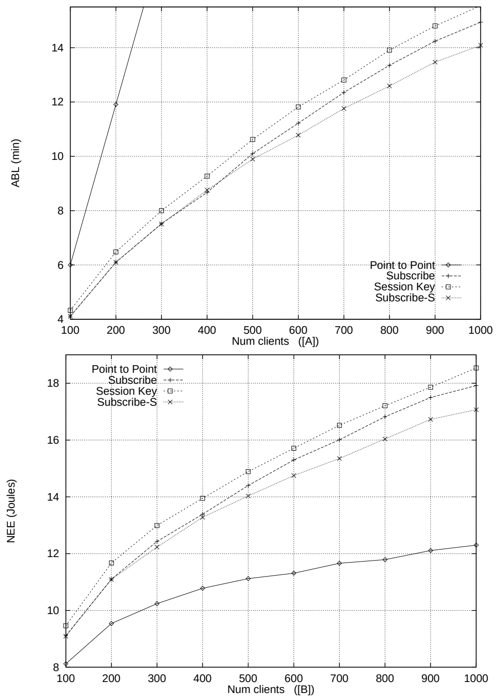

Figure 5: Sensitivity to database size: [A] ABL; [B] NEE

Figure 5 shows the effect of decreasing the size of the underlying database, which is now set to a 1000 items. Effectively, this increases the average number of subscribers for each item. This has the effect, on average, of increasing the size of key components. Essentially, a larger number of keys must be sent per item. As a result, greater savings are realized by using SubScribe^S than SubScribe, as the former uses a smaller encryption key. This is exhibited by a greater separation between the SubScribe^S and SubScribe curves.

7.3 Effect of Varying Subscription Periods

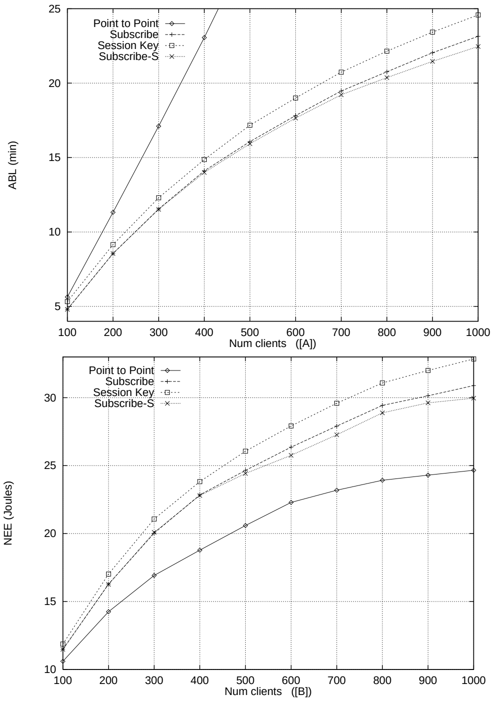

Figure 6: Sensitivity to subscription period sizes: [A] ABL; [B] NEE

Figure 6 shows the effect of increasing the mean subscription period for clients, which is now set to 90 minutes. Effectively, this increases the proportion of NODROP items, as clients stay subscribed for longer times, and consequently, fewer clients leave subscriber sets for given items. As the proportion of NODROP items goes up, the SubScribe and SubScribe^S have to broadcast fewer and fewer keys, as opposed to the SK mechanism, which must still broadcast all keys. Thus, a greater separation is realized between the SK and the the SubScribe curves. Furthermore, the ABL and NEE values have dropped for all protocols since clients are now able to download from successive broadcasts within a single subscription period. In the baseline case for higher ABL values, clients could download their data items fewer times since the subscription period was smaller.

7.4 Key Management Overhead

Finally we comment upon the key management overhead incurred by the various protocols. Specifically, we are concerned with the overhead both at the client and at the server end. It is easily seen that the client side key management cost is similar for all protocols. In all cases, the clients need to know their private key—the minimum that would be incurred in any secure application. However, the protocols differ in the additional data keys that must be remembered. In particular, in PTP, no additional keys are used. In SK, a new key is used for each broadcast and must be decrypted by each client, but can be discarded after the broadcast. In SubScribe, each client must additional remember the current data key for each subscribed data item that it is remaining subscribed to because it may be reused for the next broadcast. Finally, in SubScribe^S, each client must remember both the current data key for each subscribed data item and also the symmetric key distribution key shared by the server.

It is also seen that there is not a lot of difference in the server side key management overhead. PTP has the lowest server side overhead, where the server simply needs to lookup a public key directory for client public keys. In SK, SubScribe, and SubScribe^S, the server also needs to manage the keys with which data items are encrypted (referred to as *session keys* in SK and *data keys* in SubScribe). In SK, the server need not arrange for persistent storage of all of these keys as they are thrown away after each broadcast. In SubScribe and SubScribe^S, those data keys used to encrypt NODROP items must be remembered since they will be reused by SubScribe. As with clients, SubScribe^S has additional server side key management overhead compared to SubScribe or SK because the server also needs to manage the *update keys*, one per client. In summary, PTP has the lowest overall key management overhead, followed by SK and SubScribe, with SubScribe^S incurring the highest overhead cost.

We remark that the actual key management difference between SubScribe and SubScribe^S is somewhat subjective, and will in practice depend on factors external to the system such as whether a good public key infrastructure is in place and whether hardware support such as a smart card can help to secure users' keys, For example, if a public key infrastructure is in place, SubScribe can take advantage of it to avoid an initial registration step, while if there is not a public key infrastructure, then the initial exchange would have to occur anyway, so SubScribe^S would be preferable, since it provides slightly better performance according to the other performance measures.

8 Conclusion

In this paper, we have proposed and analyzed SubScribe,a system enabling secure data access in broadcast based information delivery environments. This research was motivated by the needs of *information commerce*, where clients pay for subscription to data items. In such environments it is necessary to permit access exclusively to data items that clients have subscribed to. Typically, encryption based schemes are used in the cryptographic community for this purpose. This problem has not been studied by the secure data management community thus far.

Using simulations, we have analyzed the performance of SubScribe and its symmetric key variant SubScribe^S under various usage parameters. We compared this performance to the performance of the point to point approach (PTP) and the session key approach (SK). Our analysis shows that both variants of SubScribe perform substantially better than PTP and somewhat better than SK in terms of average broadcast length. For normalized energy expenditure for the client, SubScribe outperforms SK, but PTP does slightly better, at the cost of significantly more work done by the server.

As ongoing work, we plan to implement a secure subscription service based on SubScribe and SubScribe^S to further evaluate their performance and other research issues that arise.

References

[1] S. Acharya, R. Alonso, M. Franklin, and S. Zdonik. Broadcast disks: Data management for asymmetric communication environments. In *Proc. ACM SIGMOD*, San Jose, California, May 1995.

[2] S. Acharya, M. Franklin, and S. Zdonik. Balancing push and pull for data broadcast. In *Proc. ACM SIGMOD*, Tucson, Arizona, May 1997.

[3] N. R. Adam and Y. Yesha. *Electronic Commerce: Current Research Issues and Applications*. Springer-Verlag, 1996.

[4] Y. Amir and A. Wool. Evaluating quorum systems over the Internet. In *Proc. 26th International Symposium on Fault-Tolerant Computing*, pages 26–35, 1996.

[5] D. Bernstein, O. Sibert, D. Van Wie, D. I. Raitt, and B. Jeapes. Information commerce: launching online content. In *Proc. International Online Information Meeting Proceedings*, Perugia, Italy, May 1995.

[6] G. Blohm, R. Parikh, E. Davis, A. R. Bhatt, and A. Ware. Information dissemination via global broadcast service (gbs). In *Proc. MILCOM 96*, McLean, VA, October 1996.

[7] C. Blundo, A. Cresti, and A. DeSantis. Space requirements for broadcast encryption. In *Proc. Advances in Cryptology - EUROCRYPT'94*, Perugia, Italy, May 1994.

[8] G. Chiou and W. Chen. Secure broadcasting using the secure lock. *IEEE Transactions on Software Engineering*, 15(8), August 1989.

[9] Primark Com. Stock information homepage. `http://www.primark.com/stock_info/stock_info.html`, 1997.

[10] PolyStor Corporation. Technical guide for lithium ion cell model icr-18650. `http://www.polystor.com/.product/tguide.htm`.

[11] A. Datta, A. Celik, J.K. Kim, D. VanderMeer, and V. Kumar. Adaptive broadcast protocols to support power conservant retrieval by mobile users. In *Proceedings of the Thirteenth International Conference on Data Engineering (ICDE 1997)*, pages 124–133, Birmingham, UK, April 1997.

[12] A. Datta, A. Celik, R. N. Wright, and A. Biliris. Secure access protocols in push-based information dissemination environment. Technical Report GOOD-TR-98-03, GOOD Group, University of Arizona, 1998.

[13] A. Datta, D. VanderMeer, A. Celik, and V. Kumar. Adaptive broadcast protocols to support efficient and energy conserving retrieval from databases in mobile computing environments. Accepted for publication in ACM TODS,, 1998.

[14] D. Denning, T. Lunt, R. Schell, M. Heckman, and W. Shockley. A multilevel relational data model. In *Proc. IEEE Symposium on Research in Security and Privacy*, Oakland, CA, April 1987.

[15] A. Fiat and M. Naor. Broadcast encryption. In *Proc. Advances in Cryptology - CRYPTO'93*, pages 480–491, Santa Barbara, CA, August 1994.

[16] Thomson Financial. Thomson financial homepage. `http://www.tfn.com`, 1997.

[17] P.A. Fishwick. *Simulation Model Design And Execution: Building Digital Worlds.* Prentice Hall, 1995.

[18] L. Gong. New protocols for third-party-based authentication and secure broadcast. In *Proc. ACM Conference on Computer and Communications Security*, Fairfax, VA, November 1994.

[19] I. S. Gopal and J. M. Jaffe. Point-to-multipoint communication over broadcast links. *IEEE Transactions on Communication*, 32(9):1034–1044, 1984.

[20] META Group. Meta group homepage. `http://www.metagroup.com`, 1997.

[21] M. Huhns and M. Singh, eds. *Readings In Agents.* Morgan Kaufmann, 1997.

[22] T. Imielinski, S. Vishwanath, and B. R. Badrinath. Data on air: Organization and access. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):353–372, May/June 1997.

[23] Motorola Inc. Meeting the challanges of a changing workforce. `http://www.mot.com/MIMS/WDG/products/datatac/prod_background.html`, 1997.

[24] I. Ingemarsson, D. T. Tang, and C. K. Wong. A conference key distribution system. *IEEE Transactions on Information Theory*, 28(5):714–720, September 1982.

[25] EIS International. `http://www.mot.com/MIMS/WDG/products/datatac/prod_background.html`, 1997.

[26] W. A. Jackson, K. M. Martin, C. M. O'Keefe, J. Pieprzyk, and R. Safavi-Naini. On sharing many secrets. In *Proc. Advances in Cryptology - ASIACRYPT'94*, Wollongong, Australia, Nov 1994.

[27] S. Jajodia and R. Sandhu. Toward a multilevel secure relational data model. In *ACM SIGMOD'91*, Denver, CO, May 1991.

[28] S. Jajodia, V. Subrahmanian, P. Samarati, and E. Bertino. A unified framework for enforcing multiple access control policies. In *ACM SIGMOD'97*, Tucson, AZ, May 1997.

[29] D. Kindred and J. M. Wing. Fast, automatic checking of security protocols. In *Proc. USENIX Workshop on Electronic Commerce*, Oakland, CA, Nov 1996.

[30] J. Levitt. Rating the push products. *InformationWEEK*, April 1997.

[31] C. Pfleeger. *Security in Computing*. Prentice Hall, 1989.

[32] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatues and public key cryptosystems. *Comm. of the ACM*, 21:120–126, Feb 1978.

[33] S. J. Shepherd. A high speed software implementation of the data encryption standard. *Computers and Security*, 14(4):349–57, 1995.

[34] *Talarian home page*. http://www.talarian.com/.

[35] *TIBCO home page*. http://www.tibco.com/.

[36] *Vitria home page*. http://www.vitria.com/.

[37] J. Weijia. Implementation of a reliable multicast protocol. *Software – Practice and Experience*, 27(7):813–850, July 1997.

[38] M. Winslett, K. Smith, and X. Qian. Formal query languages for secure relational databases. *ACM Transactions on Database Systems*, 19(4):626–662, 1994.

[39] H. Yang, J. Kim, C. Kwon, D. Won, G. S. Poo, and E. S. Seumahu. Verifiable secret sharing and multiparty protocols in distributed systems. In *Proc. IEEE Singapore International Conference on Networks and International Conference on Information Engineering*, Singapore, July 1995.

[40] S. Zdonik, M. Franklin, R. Alonso, and S. Acharya. Are "disks in the air" just pie in the sky? In *Proc. IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, California, December 1994.