Nancy-Université
Université Henri Poincaré

Henri Poincaré University – Nancy 1

Security Framework for Decentralized Shared Calendars

Master Thesis

presented and obtained publically on June 24, 2011

for obtaining the

Research Master of Henri Poincaré University – Nancy 1

(Computer Science: Services, Security and Networks)

by

Jagdish Prasad Achara

Composition of jury

	Didier Galmiche	Professor at Henri Poincaré University – Nancy 1
	Bernard Girau	Professor at Henri Poincaré University – Nancy 1
	Claude Godart	Professor at ESSTIN, Universiy of Lorraine
	Dmitry Sokolov	Associate Professor at Henri Poincaré University – Nancy 1
Supervisor :	Abdessamad Imine	Associate Professor at Nancy 2 University

Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503

Loria
Laboratoire lorrain de recherche en informatique et ses applications

Acknowledgments

First, and foremost, I would like to thank Abdessamad Imine and Michaël Rusinowitch for introducing me to the interesting field of Computer Supported Collaborative Work (CSCW) and for giving me an excellent opportunity to work under their guidance. Their knowledge and commitment with their politeness have always inspired me. They have helped me explore ideas, to critically evaluate my work, and to express my results clearly. More importantly, they have taught me how to further my skills independently. I look forward to our future collaboration and continued friendship.

I am very thankful to Tigran Avanesov for helping me throughout the project.

I would also like to express my gratitude to all of my friends for making my stay in Nancy very pleasant.

Finally, I am extremely grateful to my parents – Vimla Devi and Vidyadhar Singh – and uncle – Surender Singh – for supporting me in everything I have chosen to do and for their unbounded love and affection; my siblings – Naveen and Nirmal – for a great friendship; my nephew – Aman – for providing constant entertainment.

In loving memory of my late grandparents, Dhanne Singh and Rami Devi,
who had a sad demise in my absence when I was far away.

Contents

Chapter 1

Introduction

A shared calendar enables a user to share his calendar events only with some selected user(s) in the group to whom he wants to. In order to avoid a single point of failure, the shared calendar should not depend on a central entity. A centralized shared calendar will no more work if the central entity is down and all participating users will be affected by it instantly. A decentralized shared calendar provides far better fault tolerance as compared to its centralized counterpart. An attacker trying to stop functioning the decentralized shared calendar will only be able to do so if he can successfully perform Denial-of-Service (DoS) attack on all users of the shared calendar. On the other hand, it will take less efforts by an attacker in case of centralized shared calendar as he has to hijack only central entity on which centralized shared calendar depends. Also, Decentralized Shared Calendar provides support for Ad-Hoc networks (802.11 networks) and there is low overhead as the tasks are distributed equally among all peers. Considering the usefulness of such a decentralized shared calendar, DeSCal is proposed in [AIR11].

About DeSCal

DeSCal (abbreviation of **De**centralized **S**hared **Cal**endar for P2P and Ad-Hoc Networks) is a shared calendar application which provides users a decentralized infrastructure to share[1] their calendar events. All users are allowed to insert a new event in the DeSCal and a user who inserts a new event is the administrator/owner of this event. A user can always 'Read', 'Delete' and 'Edit' the events which he administrates/owns and he is the only user in DeSCal who can specify other user(s) with whom he would like to share the events administered by him. For fault-tolerance, availability and crash recovery, DeSCal is based on full replication of data at each user site. This implies that each user sites stores the whole copy of the shared calendar. For more in-dept detail of DeSCal, see Chapter 2 or refer to [AIR11].

Problem Statement

As the whole copy of the shared calendar is replicated at each user site in DeSCal; in absence of access control, all users can 'Read', 'Delete' and 'Edit' any shared calendar event which they would like to, even if some events are not shared with them by their administrators/owners. However, in general, it is not practical for a user to share all his calendar events with everyone in the group. For instance, a user may have some private events which he might not like to

[1] In this report, sharing of an calendar event may correspond to allow other users for any subset of <*Read*, *Delete* and *Edit*> operations on shared calendar event.

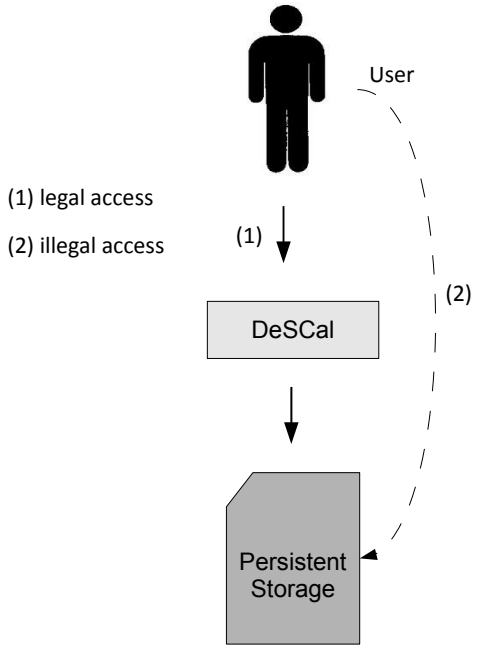

Figure 1.1: Direct access to shared calendar information in persistent storage

share with anyone in the group or he may have some secret group events which he might want to share only with some selected user(s) in the group.

To prevent unauthorized actions[2] on shared calendar events by illegal users of DeSCal, an access control mechanism is needed. DeSCal provides an access control mechanism but it deals only with 'Delete' and 'Edit' unauthorized operations *i.e.*, a user can't delete and edit the events for which he is not authorized. This access control model adopted by DeSCal is principally based on [ICR09], a flexible access control model for distributed collaborative editors which doesn't deal with 'Read' access control. However, DeSCal takes care of 'Read' access control on application level *i.e.*, a user can't read the events for which he is not authorized using DeSCal. This is accomplished by making visible to the user only the shared calendar events for which he is authorized to read. However, this is not enough to enforce 'Read' access control on shared calendar events as these events are always stored in plain-text and the whole copy of the shared calendar is replicated at each user site. A user can read these unauthorized events for 'Read' access without using DeSCal (See Figure 1.1). For example, a user can read the shared calendar events directly by reading the whole copy of the shared calendar stored locally in persistent storage (hard-disk) or reading the shared calendar events at network endpoint when they arrive from other user sites, thereby, losing/sacrificing the confidentiality of users' calendar events which is possibly an important concern for a user.

Motivation

The motivation behind this work is to make DeSCal secure by providing both confidentiality and integrity to the shared calendar events. The access control model already employed by

[2]In this report, an action may correspond to any subset of <*Read*, *Delete* and *Edit*> operations on shared calendar event.

DeSCal preserves the integrity of the replicated shared calendar events by restricting illegal users to 'Delete' and 'Edit' the events for which they are not authorized. In this report, we plan to address the issue of providing confidentiality to replicated shared calendar events by controlling the 'Read' access on shared calendar events *i.e.*, a user should only be able to read the events for which he is authorized. However, only securing the replicated shared calendar events is not enough to ensure confidentiality and integrity of shared calendar events if the network communication between users is not secure. Both providing confidentiality to users' calendar events and securing the communication between users, had been mentioned as the future work in the original paper of DeSCal [AIR11] and we plan to address these issues here.

Challenges

These security issues must be addressed in such a way that DeSCal doesn't lose its characteristic features like fault-tolerance, crash recovery, availability of data. DeSCal must be able to survive crashes, it must be fault-tolerant and data must be replicated at each user site for high availability. In DeSCal, 'Read' access control on replicated shared calendar events must be decentralized *i.e.*, there is no central server where shared calendar events can be stored and then, an access control policy is enforced on this centrally stored data. Decentralization leads to new challenges like self-organization, service availability and security. Moreover, in DeSCal, the group of users is dynamic *i.e.*, users leave or join the group in an arbitrary manner at any point of time.

Contribution

Our contribution consists of (i) proposing a required security framework for DeSCal and (ii) implementing it on top of iPhone OS implementation of DeSCal [AIR11]. The proposed security framework provides (i) confidentiality to replicated shared calendar events and (ii) secures the communication between users in DeSCal. It is achieved using standard cryptographic protocols in the field of computer security. This proposed security framework preserves all characteristic features of DeSCal.

This report is organized as follows:

In Chapter 2, we detail the desired features of a shared calendar and briefly present a Decentralized Shared Calendar (DeSCal) [AIR11]. Chapter 3 establishes the security requirements of DeSCal whereas Chapter 4 presents the state of the art of securing replicated data in a decentralized distributed context. Security framework is proposed in Chapter 5 and Chapter 6 describes its implementation on top of iPhone OS implementation of DeSCal. In the end, we present the summary of our work and possible directions of future work.

Chapter 2

Background

This chapter essentially brings forward what a reader has to know before proceeding to read this report any further. It investigates the desired features of a shared calendar, and then, briefly introduces DeSCal [AIR11]. In case a reader already knows the usefulness of a decentralized shared calendar and is familiar about DeSCal, this chapter can exceptionally be skipped solely based on reader's own judgment (or decision). In any case, we strongly recommend reading this chapter to everyone as DeSCal is presented here in a better, different and compact manner.

2.1 Desired Features of Shared Calendars

The shared calendar must not depend on a third party.

Shared calendars based on a third party central server, for instance, Google Calendar [GCa], prevents users to create a dynamic Ad-Hoc group as users always have to communicate through this central server. Moreover, there is no direct communication between users and they need to have a constant connection to this third party central servers (e.g. an Internet connection to connect to Google Servers to use Google Calendar). Also, using third-party central server to store the shared calendar information, reduces availability of the data as the data has to be fetched from or stored at remote central server when modified by users. Above all, using a third party for sharing calendar events may also lead to sacrificing the confidentiality of these users' calendar events which is possibly an important concern for a user. See Figure 2.1 to better understand the disadvantages behind using a third party to store the shared calendar information.

An access control mechanism is needed on shared calendar events.

It may not always be appropriate for a user to share all his calendar events with other users in the group. A user may have some secret events which he would like to share only with some selected user(s) in the group, not with everyone. To deal with this aspect of shared calendar, there must be some access control mechanism on shared calendar events. Moreover, access control on shared calendar events must be dynamic, *i.e.* users should be able to change the access rights on their shared calendar events at any point of time after the creation of an event (unlike the case where a user specifies access control for an event only at the time of creation or insertion of an event).

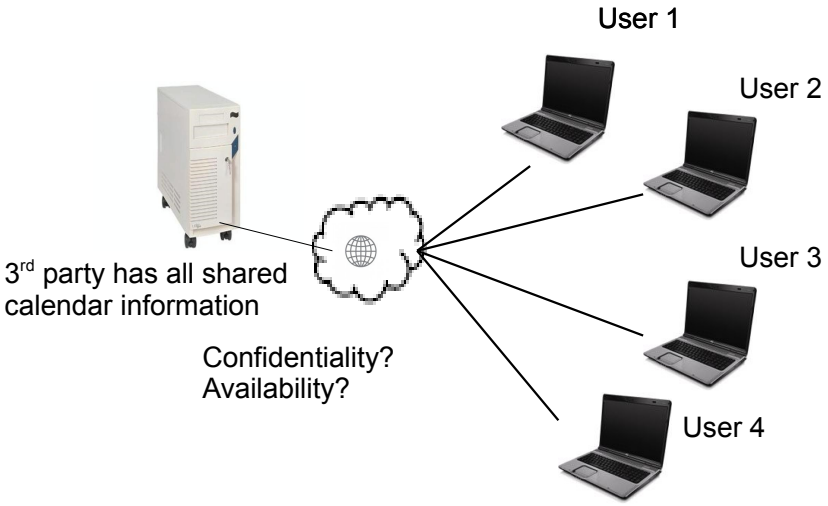

Figure 2.1: Disadvantages behind using a third party to store the shared calendar information

Eventual Consistency and High Concurrency

Eventual consistency helps concurrency and parallelism as there is no need to wait for all of the copies to be synchronized when updating shared calendar. Any number of users should be able to concurrently modify the shared calendar (by inserting, deleting and editing the events in the shared calendar) but users must eventually be able to see a converged view of all replicated shared calendar copies. Eventual consistency of the replicas is desired; otherwise, diverged replicas may result into a wrong insertion, deletion and edition of the events in the replicated shared calendar copies.

Scalability

The shared calendar must be dynamic in the sense that users may join or leave the application at any point of time during its runtime.

High responsiveness

The shared calendar must be as responsive as a personal calendar i.e. users should have an illusion that they are alone while using the shared calendar.

2.2 DeSCal

A decentralized shared calendar, **DeSCal**, complying with all these desired features of a shared calendar has been proposed in [AIR11].

2.2.1 Is DeSCal the first one of its kind?

Shared calendars are common nowadays but none of them is decentralized and self-configurable except DeSCal. Google Calendar [GCa] is a shared calendar application by Google but it needs

a constant connection to Google servers and a prior registration for its shared calendar service by each user. Consequently, it can't be used over local Wi-Fi networks or over blue-tooth in an organization where mobile users can leave or join the group in an Ad-Hoc manner. This gives DeSCal an edge over its centralized counterparts.

A user can use mobile version of DeSCal and join or leave the group in an Ad-Hoc manner. There exists mobile version of Google Calendar which is made for small screen and also, mobile phone's built-in calendars can be synchronized with Google Calendar when users are away from their desk but the need to depend on a third party (i.e. a constant connection to Google Servers) never goes away.

Zimbra platform calendar application [Zim] enables users to share their events with other users in a group but again, this is centralized and a server needs to be run before using this application. While using Zimbra platform calendar application, one can run his own server in an organization without depending on a third party but this central server becomes a single point of failure for the shared calendar. Furthermore, centralized server prevents users to create Ad-Hoc group as users need prior knowledge of central server to run the application.

Both Google Calendar and Zimbra platform calendar application use CalDAV (Calendar extensions for Distributed Authoring and Versioning) [Cal] which is an Internet standard allowing a client to access scheduling information on a remote central server. To the best of our knowledge, all other shared calendars depends on a central entity to run and manage itself.

2.2.2 A Deployment Scenario of DeSCal

A research team in a research organization usually consists of a scientific and administrative leader, other members of the team and an administrative assistant. All members of the team can run an instance of DeSCal to keep track of their personal events and also, to share some group events with others. They can hide their personal events by not sharing these events with others. For group events involving two or more persons, one can create the event and share it with other members who are concerned by it. If few members of the team are in a group meeting, others can know this by just having a look on DeSCal. In addition, it is more intuitive for a team leader to share some administrative events with delete and/or edit right with the administrative assistant of the team so that extra hassle of communication can be avoided to deal with administrative tasks.

At the same time, we agree with the fact that this can also be easily done using a centralized shared calendar like Google Calendar but members of this research team can find DeSCal more appropriate if any one or all of the following possible scenarios are true:

- The central entity in the centralized shared calendar is not owned by the organization itself and also, this organization has no connectivity with the outside network of a third party on which centralized shared calendar depends. For example, this research team can't use Google Calendar if they don't have Internet connection as it is managed by a third party.
- Members of the team want to keep their calendar events confidential i.e. they don't want to disclose their calendar events with this third party (for instance, Google in case they use Google Calendar) who provides shared calendar service.

- A team member meets someone while going for lunch or for a coffee and he wants to share some events with this person. This can be done instantly without any overhead like registering for shared calendar service of a third party; just by allowing other person to join the group and sharing only some specific events which this team member wants to share.

2.2.3 Ingredients of DeSCal

Here, we briefly present the elementary entities of DeSCal as it is crucial to understand the proposed security framework.

Shared Calendar

The shared calendar can be modeled by the list abstract data type where each element of the list is an event [ICR09].

Definition 1. ***[Cooperative Operations].*** *The shared calendar state can be altered by the following set of cooperative operations: (i) Ins(p, e) where p is the insertion position in the shared calendar and e the event to be inserted at p; (ii) Del(p, e) which deletes the event e at position p; (iii) Up(p, e, $e^{'}$) which replaces the event e at position p by the new event $e^{'}$.*

Policy

The access control model in DeSCal is based on *authorization policy* or *access control policy*[3] where **each user defines a policy to specify access control rules for the events created by him**. A policy is a list of authorizations or rules; sometimes, also called *authorization list*. A policy specifies the operations a user can execute on shared calendar events. Three sets are used for specifying policies, namely:

1. S is the set of *subjects* where a subject is a user.
2. O is the set of *objects* where an object is an event.
3. R is the set of *access rights*. Each right is associated with an operation that a user can perform on shared calendar events. Thus, we consider the right of reading an event (rR), inserting an event(iR), deleting an event (dR) and updating an event (uR).

As each user is authorized to insert an event in the shared calendar, the insertion right (iR) is not dealt and so, DeSCal must deal with dynamic changes of rR, dR and uR rights.

Definition 2. ***[Policy].*** *A policy is a function that maps a set of subjects and a set of objects to a set of signed rights. We denote this function by $P : \mathcal{P}(S) \times \mathcal{P}(O) \rightarrow \mathcal{P}(R) \times \{+,-\}$, where $\mathcal{P}(S)$, $\mathcal{P}(O)$ and $\mathcal{P}(R)$ are the power sets of subjects, objects and rights respectively. The sign "+" represents a right attribution and the sign "–" represents a right revocation.*

A policy P is represented as an indexed list of *authorizations* (or *rules*). Each authorization P_i is a quadruple $\langle S_i, O_i, R_i, \omega_i \rangle$ where $S_i \subseteq$S, $O_i \subseteq$O, $R_i \subseteq$R and $\omega_i \in \{+,-\}$. An authorization is said *positive* (resp. *negative*) when $\omega = +$ (resp. $\omega = -$). Negative authorizations are just used to accelerate the checking process. When a user wants to perform an operation, DeSCal needs

[3]In the rest of the report, we only write 'policy' instead of writing 'access control policy' or 'authorization policy' each time.

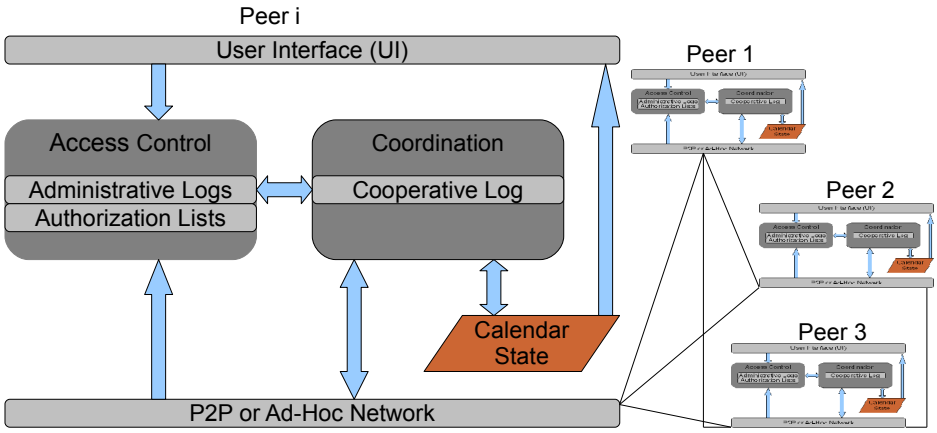

Figure 2.2: DeSCal design modules

to check whether the user is allowed to perform this operation or not. DeSCal uses a first-match semantics for this: when an operation o is generated, it checks o against its authorizations one by one, starting from the first authorization and stopping when it reaches the first authorization l that matches o. If no matching authorization is found, o is rejected, *i.e.* that operation can't be executed on the shared calendar.

Definition 3. ***[Administrative Operations].*** *The state of a policy is represented by a list of authorizations $\langle P \rangle$. A user can alter the state of his policy by the following set of administrative operations: AddAuth(p, l)/DelAuth(p, l) to add/remove authorization l at position p. An administrative operation r is called restrictive iff r = AddAuth(p, l) and l is negative or r = DelAuth(p, l).*

2.2.4 DeSCal modules

The design of DeSCal is composed of four well-separated conceptual modules described below (See Figure 2.2):

Coordination

The job of coordination module is to handle the concurrent updates on the shared calendar by different users. This has to be done in a decentralized and scalable manner, *i.e.* without requiring a central server and where users can leave or join the group at any point of time. It is this module which is responsible for ensuring the same converged and consistent copy of the shared calendar at each participating node in all cases. It directly interacts with the local copy of the shared calendar and thereby, is responsible for maintaining its consistency. This module is based on [Imi09], a coordination module for distributed collaborative editors.

Access Control

This module is to control access on the shared calendar events so that a user is only able to access the events for which he is authorized. Also, the access control mechanism provided by this module is dynamic, decentralized and scalable. Here, dynamic access control on calendar events means a user can change access control for his shared calendar events at any point of time after the creation of an event.

P2P/Ad-Hoc Network

The role of this module is to maintain a local knowledge of the network infrastructure. It's the responsibility of this layer to provide Peer-to-Peer distributed architecture services to DeSCal for any kind of network like Wireless Ad-Hoc networks, Short range communication (e.g. bluetooth), LAN, Internet or Managed infrastructure Wireless LAN.

User Interface

It enables users to take actions on the shared calendar. However, this module can't directly change the state of the shared calendar without interacting with the coordination and access control module. An action taken by the user on this module has to pass through access control module. If this action is authorized by access control module, it is passed to coordination module to change the state of the shared calendar and to deal with consistency issues.

2.2.5 DeSCal Workflow

Below, we present the sequence of major steps taken by DeSCal to better understand it while hiding the complexity behind these steps:

1. When a user manipulates an event in the local copy of the shared calendar by generating a cooperative operation, this operation will be granted or denied by only checking the local copy of the policy of the administrator of that particular event.

2. Once granted and executed, the local cooperative operation is then broad-casted to other users. On reception of this operation at remote user sites, a user has to check whether the remote operation is authorized with respect to his locally stored *admin log*[4] of the event's administrator before executing them on his local copy of the shared calendar.

3. When a user modifies his local policy by adding or removing authorizations, he sends these modifications, *i.e.* administrative operations, to other users in order to update their local copies of the policy.

2.2.6 What does a DeSCal user site need to store?

In DeSCal, all users are authorized to insert a new event and a user who inserts an event is the administrator of that event. An administrator can take any action on the shared calendar event created by him and he is the only one who can decide if some other user in the group is allowed to take these actions on this shared calendar event. An administrator exerts this authority on his shared calendar events by specifying authorizations or rules in his policy for the events administered by him. As each user is the administrator of the events created by him, there are as many policies as the number of users. Users can modify an event in shared calendar with respect to the locally stored policy of the administrator of that event.

Shared Calendar and Cooperative Log. DeSCal keeps a copy of the shared calendar at each user site to improve performance. Users can perform updates on his local copy of the shared calendar independently and then, these locally executed updates are transmitted to other users in the group. One has to note here that reception of these locally executed updates at other user

[4]Short form of administrative log. Administrative log of user u_1 stores all administrative requests generated by him.

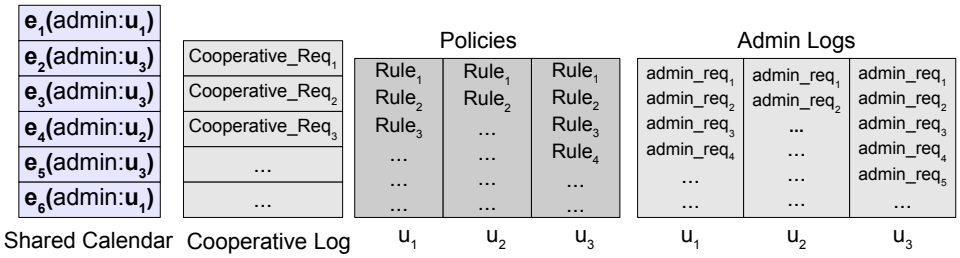

Figure 2.3: Storage at u1's local site

site is not guaranteed to be in the same order as they are executed locally at their corresponding local sites because of network latency variation, computer resources etc. Therefore, DeSCal needs a decentralized mechanism which can handle the coordination of concurrent updates by different users on shared calendar. The mechanism used in [AIR11] keeps track of both local and remote calendar update requests by storing them in a log called *cooperative log*. This log helps in maintaining the consistency of the shared calendar and it needs to be stored at each user site.

Policies and Admin Logs. To deal with latency and dynamic access changes, it uses an optimistic access control technique (inspired from [ICR09]) in such a way that enforcement of authorizations is retroactive. Access control is the ability to grant or deny the manipulation of information by someone. A data structure (*policy* in our case) is used to store all access rights. This *policy* is checked whenever the controlling-access is started. We store the *policy* at each user site because high responsiveness can be lost if every update must be authorized by some authorization coming from a distant central server. DeSCal follows the access control model described in [ICR09] but this access control model doesn't satisfy all the requirements of DeSCal. The model proposed in [ICR09] is single-administrator whereas in DeSCal, each user is the administrator of the events created by him. The single administrator model, proposed in [ICR09], keeps a copy of the *admin log* containing all administrative requests and *policy* at each user site. DeSCal extends this model to make it a multi-administrator model where each user has his own 1) *policy* for the events created by him and 2) *admin log* to store his administrative requests. Eventually, this requires each user to keep a copy of the *policy* and *admin log* of all other users in the group.

To sum up, we illustrate what a user site stores by presenting an example scenario (See Figure 2.3). Suppose, there are three users in DeSCal at a given time t: u_1, u_2 and u_3, and they insert concurrently two, one and three events respectively. After insertion of these events, each user will have the same copy of the shared calendar. In this case, events e_1 and e_6 are inserted by user u_1; e_2, e_3 and e_5 are inserted by u_3 and e_4 is inserted by u_2. Cooperative log at each user site will contain all the cooperative requests generated locally and received from remote user sites. User u_1 will store the policies and admin logs of each user including himself (*i.e.*, u_1, u_2 and u_3) and admin log stores all the administrative requests generated by the corresponding user. In this case, u_1 will store the following at his local site : *policies* and *admin log* of u_1, u_2 and u_3, whole copy of the *shared calendar* and his *cooperative log*.

Chapter 3

Security Requirements of DeSCal

Securing DeSCal had been left as the future work in the original paper of the DeSCal [AIR11] and because of absence of security measures, DeSCal remains vulnerable to all kind of attacks. Here, in this chapter, we list what is necessary to make DeSCal secure while at the same time, not loosing its unique features like fault-tolerance, crash recovery, availability, dynamic access control. Our focus is primarily on analyzing and precisely establish the security requirements for DeSCal.

Mainly, below two tasks were left as the future work in the original paper of the DeSCal [AIR11] and we will deal with these in this report.

3.1 Providing confidentiality to replicated shared calendar events

The whole copy of the shared calendar is replicated at each user site in DeSCal but a user should be able to read, delete or edit only the events for which he is authorized. For shared calendar events not to be deleted/edited by unauthorized users, DeSCal [AIR11] already employs an access control module, which essentially ensures the integrity of the replicated shared calendar events. However, this access control model can't prevent users to read the shared calendar events for which they are not authorized. It is of no use to provide confidentiality to shared calendar events as in DeSCal, the whole shared calendar is replicated at each user site in plain-text. As a matter of fact, a user can read this locally stored shared calendar state directly without using the application (or outside the scope of the application) even if DeSCal generates a view of 'Read' access authorized calendar events for the user to hide the unauthorized shared calendar events for 'Read' access. This leads to loosing the confidentiality of shared calendar events. So, here, our primary goal is to provide confidentiality to shared calendar events *i.e.*, only authorized users can read the shared calendar events. We highlight here that we don't deal with the integrity of shared calendar events as it has been already dealt by access control module employed in DeSCal.

Below, we discuss informally what additional data needs to be secured apart from the replicated shared calendar and why this data needs to be secured. As it is clearly evident from the fact that DeSCal is pure P2P, we need to store the copy of the shared calendar at each user site. In addition to storing the shared calendar, as described in Chapter 2, a user in DeSCal also needs to store following four entities at each user site:

- Shared Calendar

- Policies of each user
- Admin logs of each user and
- Cooperative log

Storing local copy of all these data leads to high availability of data, thereby, also increasing the responsiveness and performance of the DeSCal. While providing confidentiality to shared calendar events, care must be taken as a cooperative log stores all local and remote cooperative requests. It requires us to provide confidentiality to shared calendar events stored as cooperative operations in cooperative requests. Policies and admin logs of all other users stored at each user site don't need to be secured in DeSCal as they (policies and admin logs) contain the access control information on shared calendar events. Reading these copies of the policies and admin logs can't help a user to get any significant information about shared calendar events except the unique event ids of events. Even if a user changes these locally stored copies of admin logs and policies of other users, he will end up having his own shared calendar copy diverged from other shared calendar copies stored at various other user sites. Other users won't be affected by it because they will simply ignore his unauthorized actions on shared calendar events once they will be received at their corresponding local sites. On reception of his unauthorized actions at remote user sites, these actions can be checked against remote users' local copies of the polices and admin logs; and can be rejected if not allowed according to their locally stored policies or admin logs.

3.2 Securing the communication between users

Besides confidentiality to replicated shared calendar events, DeSCal [AIR11] doesn't provide any security measures to secure the messages exchanged among users. The network communication between users must be secured with respect to all basic security properties like confidentiality, integrity, non-repudiation, replay attack. It is an important fact to point out that in DeSCal, a number of users form a group to participate in the shared calendar but the group communication security mechanisms are not suitable for DeSCal. At first glance, it might seem a little bit awkward but in DeSCal, the messages broadcast are not intended to be read by everyone in the group; as opposed to DeSCal, it is actually the case in group communication. It is due to the fact that, unlike in group communication where all users are equally privileged, users in DeSCal have different rights. In DeSCal, for each event, only a subset of users in the group are authorized to read. A message broadcast to all users in DeSCal should not be read by the users who are not authorized to read it while it is in network transmission. For example, if a user u_1 wants only user u_2 to read his event, then, user u_3 (also, part of the group) should not be able to read this event. Therefore, when this network message is broadcast to all the users in the group through a communication channel, it needs to be secured in such a way that user u_3 (who belongs to this group) should also not be able to access it. Here, in DeSCal, a broadcast network message has to be secured from unauthorized users in shared calendar group of DeSCal as well as outsiders while it is in transit.

Chapter 4

State of the art

Today, P2P systems are rapidly growing in use because of their inherent nature to provide 1) scalability, 2) equal participation of peers and 3) fault-tolerance. These systems are fault-tolerant as 1) there is no single point of failure and 2) they replicate the data at each participating node. However, with the absence of a central authority, security of 1) replicated data and 2) messages exchanged between peers, is a challenging task. Further, P2P systems can be classified broadly in three groups depending on their purpose of use: 1) Content distribution 2) Distributed computation and 3) Collaborative application. Likewise, security requirements for these different classes also differ from each other. We start by doing a literature survey of security aspects in collaborative applications and of course, in other decentralized shared calendars which is a variety of decentralized collaborative environment.

4.1 Security mechanisms adopted in other shared calendars and decentralized collaborative environments.

To the best of our knowledge, there doesn't exist other decentralized shared calendars where a literature survey of their security aspects can be done. Besides, securing a centralized shared calendar is totally different and is not relevant as it involves only securing the data stored at a single central entity. Though, as a matter of fact, DeSCal is fundamentally a type of decentralized collaborative environment where a group of users take part and share their calendar events by working on the same copy of the replicated shared calendar. Consequently, our first natural direction of literature survey is the field of decentralized collaborative environment. We explored few works [AD06], [Liu08], [LWR09], [AIS+05], [Ada06] and [GJO+06] for access control in the field of decentralized collaborative environment. However, we found out that all these works are based on trust i.e. they use trust-based access control (TBAC). Nonetheless, trust-based access control is not suitable to DeSCal as the requirements of DeSCal include immediate enforcement of user's access control policy. In DeSCal, once a user revokes or shares his event(s) with some selected user(s), this must be followed/performed by the application strictly and immediately. DeSCal can't tolerate the time taken to build the trust between users which is normally the case in trust-based access control systems. Moreover, trust-based access control mechanisms enforce access control based on trust and not according to the user's access control policy.

We again remind here that DeSCal already employs an access control mechanism for preserving the integrity of the shared calendar (by restricting unauthorized 'Delete' and 'Edit' operations on shared calendar events) and this access control mechanism is based on strict access control

according to user's policy and not based on trust. However, this access control mechanism doesn't deal with 'Read' access control on shared calendar events loosing the confidentiality of users' calendar events and here, in this report, our primary goal is to provide confidentiality to users' calendar events.

4.2 Securing replicated data.

As outlined above, trust-based access control is not suitable for DeSCal. Regardless, we don't limit ourselves only to works done for access control in the field of collaborative environments. We realize that the problem of providing confidentiality to replicated data seamlessly exists in other kinds of decentralized distributed systems which are based on replication of data. In this progression, first, we investigate the revolutionary work done by Herlihy and Tyger in their paper [HT88] for securing the replicated data. This paper describes a mechanism to make replicated data secure using 1) Quorum consensus replication [Her86] and 2) Shamir's secret sharing algorithm [Sha79]. Here, securing replicated data means preserving its confidentiality and integrity. Authors describe protocols employing private and public key encryption to preserve the confidentiality (secrecy) and integrity of the data. However, this work preserves the confidentiality of the data only against a passive adversary and not against an active adversary while in DeSCal, confidentiality of the data must be preserved against an active adversary. Here, active adversary means the owner of a user site where data is stored whereas passive adversary can be anyone else who tries to read/modify the contents of the data. One has to note here that, in [HT88], encrypted data is stored in replicated form in a quorum, while the key used for encrypting this data is stored using Shamir's secret sharing scheme [Sha79].

Policy-based access control for weakly consistent replication systems is proposed in [WRT09]. Authors describe the design and implementation of this access control model within Cimbiosys replication framework [RRT+09]. They describe this access control system for replicated data by presenting a real-world scenario. In this access control model, replicas have a version number associated with each update of the replica. Access control on these replicas is limited in the sense that revocation claims are applied only to those versions newer than the associated version vector. For example, if read right is revoked at version v for a peer p by the owner, p will not be able to read newer versions but still, he will be able to read v or versions older than v. However, in DeSCal, if an administrator revokes 'Read' right for his events at time t for a peer p, then, even concurrent modifications at the same time t at other user sites must not be read by user p *i.e.* in DeSCal, administrative operations are given more priority than cooperative operations. Not to mention, in the proposed access control model in [WRT09], data is weakly consistent because consistency of replicated data depends on time interval between two synchronization operations provided by replication framework whereas DeSCal is based on eventual consistency.

In an another work [PTHCR08], secure content access and replication in pure P2P networks, security aspects required in content sharing application are dealt with and a new security mechanism is proposed which suits best to content sharing applications.. Authors deal with the security aspects like content authentication and content integrity checking in content sharing applications. Security mechanism is provided in a distributed fashion (i.e. without requiring a central authority) using Pathak and Iftode's protocol [PI06] for public key authentication. The authentication of public key is very important to associate a user with a public key i.e., public-key authentication ensures that a user who claims to have a public-private key pair actually

possesses the private key corresponding to that public key. The proposed protocol is based on associating content with security labels. Appropriate user clearances are required to access this labeled content. It is similar to PGP in the sense that content and authorization certificates are issued by the users. However, as opposed to PGP, it doesn't rely on certificate directories for the distribution of certificates. In any case, this mechanism deals only with the integrity of data by preventing malicious users changing the content of the published data and confidentiality of the data is not dealt.

4.3 Secrecy by splitting.

For security and fault-tolerance, another set of works mainly rely on either Shamir Secret Sharing Algorithm [Sha79] or Rabin's Information Dispersal Algorithm (IDA) [Rab89]. A different (and somewhat less efficient) perfect secret sharing scheme [Bla79] was also proposed by Blakley in the same year as Shamir's. However, Shamir's scheme is used more frequently than Blakley's in the works based on perfect secret sharing schemes (possibly because of its efficiency advantage). As opposed to this, in our opinion, both Shamir's and Blakley's schemes can be used interchangeably when it comes to use a perfect secret sharing scheme (if efficiency factor can be ignored). Below, we briefly describe Shamir's secret sharing and Rabin's information dispersal schemes respectively for readability and completeness.

Shamir Secret Sharing Algorithm. It divides data D into n pieces in such a way that D is easily reconstructible from any k pieces, but even complete knowledge of k-1 pieces reveals absolutely no information about D. It can be used to divide the secret data D (preferably, a cryptographic key) into n pieces where pieces are stored in a distributed fashion. Secrecy of the data is preserved due to the fact that any lesser than k pieces of data don't reveal any information about the secret data D. Likewise, fault-tolerance is achieved because of the fact that any k out of n pieces can be used to reconstruct the data D.

Rabin's Information Dispersal Algorithm (IDA). It breaks a file F of length L into n pieces, each of length L/m, so that every m pieces suffice for reconstructing F. Dispersal and reconstruction are computationally efficient. The sum of the lengths is (n/m) $\times$ L. Since n/m can be chosen to be close to 1, it is space efficient. It has numerous applications to secure and reliable storage of information.

Shamir Secret Sharing Algorithm can only be used for the data which is (or can be made) a number, thereby, having an upper limit on the length of the data. It was originally designed for the construction of robust key management schemes for cryptographic systems that can function securely and reliably even when misfortune destroy half the pieces and security breaches expose all but one of the remaining pieces. As opposed to this, Rabin's IDA can be used for any arbitrary length of data. Shamir's and Rabin's techniques achieve a different level of security with different performance and storage requirements. If the original file is b bytes in size and the file is to be divided into n pieces such that any k pieces suffice to reconstruct the file, Shamir's scheme requires a total of $n \times b$ bytes, while Rabin's requires $(n \times b)/k$. Shamir's requires more computation as well. To compensate for the extra storage and computation, Shamir's scheme is more secure, achieving information theoretic security. Rabin's security is far less, and would be unacceptable in many environments. However, in 1993, Krawczyk proposed a blending of Rabin and Shamir for improved security, by encrypting the data with a key-based encryption

algorithm, and then dispersing the encrypted data with an IDA and the key with a secret sharing scheme [Kra94]. This is called *Secret Sharing Made Short (SSMS).*

Below are some works which provide secrecy to the data by either splitting the data itself or the cryptographic key which is previously used to encrypt this data.

In this context, we start by investigating an approach for fault tolerant and secure data storage in collaborative work environments [SB05]. As, in collaborative work environments, it can be expected that there will be changes in the list of users authorized to read or update the sensitive data. When using the traditional approach, changes in the access list will require re-encrypting the stored data with a new cryptographic key, which may be cumbersome. For fine grained access list management, each file or document stored would require a unique key. The number of keys could then become large and unmanageable. To avoid such expensive operations during changes in the access list, authors of [SB05] propose to store the data itself using secret sharing techniques. However, it is not clear for us by reading this paper that how do authors really enforce these access list changes on the stored data (for example, addition of a new user in access list). Moreover, unlike in DeSCal, they store data at distributed data storage servers whereas in DeSCal, data must be fully replicated at each user site.

In [AAEM09], authors aim to outsource data in such a way that outsourced data is guaranteed to be secure. In this paper, authors describe scalable privacy preserving algorithms for data outsourcing. Instead of encryption, authors use distribution of data on multiple data provider sites and information theoretically proven Shamir's secret sharing algorithm as the basis for privacy preserving outsourcing. However, **Shamir's shared secret method can't be used in DeSCal** because:

- Firstly, as already explained, Shamir's secret sharing scheme applies only to the data which is (or can be made) a number, thereby, restricting the size of the secret data on which it can applied. It is frequently applied to provide secrecy to cryptographic keys which are generally small in size.
- Secondly, in DeSCal, users are dynamic *i.e.* users may join or leave the group at any point of time and hence, requiring the threshold of Shamir's shared secret method to be changed accordingly. Moreover, this change in threshold must be performed without requiring the dealer (dealer is the one who breaks the secret data into pieces and distributes to others) because in DeSCal, an administrator of the event might not be present in the group when a new user joins the group or an existing user leaves the group.

Several works [TW], [SWP04], [BCSV94], [MMT01] exists in literature to change the threshold parameter of the Shamir's shared secret method with or without requiring the dealer. Nevertheless, [TW] and [SWP04] only investigates the problem of increasing (not decreasing) the threshold parameter of the Shamir's scheme. Furthermore, in actual, [TW] assumes that threshold t is beyond 160 which is, in any case, not practical for DeSCal as it is not practical to have number of users 160 or more in a shared calendar group of DeSCal. [MMT01] again addresses the problem of changing the threshold parameter of Shamir's secret sharing scheme along with extra verification, in which the combiner can verify whether the pooled shares are correct or not and also, the participants can verify whether the share given by the dealer is correct or not. But, in this scheme, the threshold can be changed only plural times to the values determined in advance.

[BCSV94] is specially designed for future threshold modification and is based on advanced share technique. Per contra, dynamic secret sharing schemes based on advanced share techniques can't be used in DeSCal as the group of users is extremely dynamic and can't be predicted at a certain point of time (even the number of users in DeSCal can't be predicted in advance). Apart from the works done for changing the threshold of Shamir's secret sharing method, in [WXYX07], a new dynamic threshold secret sharing scheme from Bilinear Maps is proposed. However, it needs a public *bulletin board*, under the control of the dealer, to publish auxiliary information. In [Cac95], an another work which uses *bulletin board*, a new construction for computationally secure secret sharing schemes with general access structures is proposed. This scheme provides the capability to share multiple secrets and to dynamically add participants on-line, without having to re-distribute new shares secretly to the current participants.

In [GKLL], a system, *vanish* is proposed to increase the privacy of data by self-destructing it after a user-specified time. This system seeks to protect the privacy of past, archived data – such as copies of emails maintained by an email provider - against accidental, malicious and legal attacks. Specifically, they wish to ensure that all copies of certain data become unreadable after a user-specified time. It is a new idea to destroy unwanted data after a specific time-limit and helps increase data privacy. They combine shamir's secret key concept and DHTs in a nice way leading to develop this system. For security of data without using encryption techniques, as described above, a lot of works split and store the data across various physical locations. POTSHARDS [SG07] is a system designed to provide secure long-term archival storage to address the new challenges and new security threats posed by archives that must securely preserve data for decades or longer. This archival storage system, POTSHARDS, provides long-term security for data with very long lifetimes without using encryption. Secrecy is achieved by using provably secure secret splitting and spreading the resulting shares across separately-managed archives. However, shared secret splitting methods can't be used in DeSCal to share a secret as the group of users in DeScal is highly dynamic.

Dispersing data across multiple sites yields a variety of obvious benefits, such as availability, proximity and reliability. Less obviously, it enables security to be achieved without relying on encryption keys. Standard approaches to dispersal either achieve very high security with correspondingly high computational and storage costs, or low security with lower costs. AONT-RS [RP11], a new dispersal scheme is proposed which blends an *All-Or-Nothing Transform* [Riv97] with *Reed-Solomon coding* to achieve hight security with low computational and storage costs. It uses Rabin's IDA and enriches it in two ways: first, by employing a variant of Rivest's *All-or-Nothing Transform* as a preprocessing pass over the data and second, by employing a *systematic* erasure code instead of *non-systematic* one. Unlike SSMS [Kra94], it enriches the security of Rabin's IDA without secret sharing. But again, we can't use Rabin's IDA in DeSCal due to below reasons. In DeSCal, when a new user joins the group, the administrator of the event must send him his share but at the time of joining the group, the administrator may not be present in the group. Also, a new user must contact all the existing users in the group to get his shares of the shared calendar events at the time of joining the group which is not practical.

In some other work [JAV08], a privacy service for DHTs is proposed by applying the principles of Hippocratic database to P2P systems to enforce purpose-based privacy which prevents privacy violation by prohibiting malicious data access. However, data is not replicated and is stored only at a single data provider site.

Chapter 5

Proposed Security Framework

In this chapter, our primary goal is to propose a decentralized security mechanism which can fulfill all security requirements of DeSCal. The proposed security framework satisfies all the security requirements of DeSCal while at the same time, not loosing DeSCal's characteristic features. This novel security framework makes use of well-established cryptographic protocols in the field of computer security.

Things to be taken care of while proposing the security framework for DeSCal (Security Framework Design Requirements).

Both providing confidentiality to a user's calendar events and full replication insists DeSCal to store the shared calendar events in some non-readable (encrypted) form where a user can read only the events for which he is authorized. However, just storing these shared calendar events in a non-readable form doesn't make any sense. In the original paper of DeSCal [AIR11], the motivation behind full replication was to increase the availability of the data, ensuring recovery of the data in case of crash and to make DeSCal fault-tolerant. Therefore, it is very important to store this non-readable in such a way that DeSCal doesn't loose it's all above mentioned salient features.

Fault-tolerance is the ability of DeSCal to continue operating properly when a user site crashes. In addition, DeSCal has the ability to survive the crash of a user site i.e. if a user site is crashed, this user can join the group and participate again in the same manner as it was participating before the crash by restoring the whole state of the shared calendar. A term 'maintainability' was also coined for this later aspect of DeSCal. Availability of data ensures that the shared calendar events are readily available to a user site when they are needed i.e. if a user wants to access the shared calendar events for any subset of 'Read', 'Delete' or 'Edit' operations, these shared calendar events must be retrieved as quickly as possible. For immediate retrieval to increase the availability of the data, these shared calendar events must be stored at each user site. Storing these shared calendar events at each user site ensures that these events are present and ready for use all the time. Furthermore, for these stored shared calendar events to be available all the time, the security framework used to provide confidentiality must also be functioning correctly all the time. Ensuring availability also involves preventing denial-of-service attacks. Therefore, while proposing a security framework ('Read' access control on shared calendar events and securing the communication) for DeSCal, one must take extremely care that these salient features of DeSCal are retained.

Here, we would also like to point out that security framework must essentially be designed on top of DeSCal [AIR11] where coordination and access control models already employed by DeSCal are not changed substantially. They must remain as intact as possible and can have merely very minor modifications. We remind here that these coordination and access control models used in DeSCal are mainly based on coordination and access control models proposed in [Imi09] and [ICR09] for decentralized distributed collaborative editors respectively.

DeSCal is based on broadcast group communication *i.e.*, it doesn't need to send specific messages to different users. And, coordination and access control modules already employed by DeSCal are designed keeping in mind this fact. Thereupon, another design requirement of security framework for DeSCal includes preserving this broadcast group communication.

In the end, we would like to emphasize the fact that providing the required security mechanisms for DeSCal in such an environment is very challenging task and has never been dealt by the research community in this field so far to the best of our knowledge. The challenges are:

1. 'Read' access control must be decentralized *i.e.*, there is no central server where we can store the data and then, enforce an access control policy on this centrally stored data.
2. DeSCal must be able to survive crashes; it must be fault-tolerant and data must be replicated at each user site for high availability
3. Users in DeSCal are dynamic *i.e.* a user can leave or join the group at any point of time.
4. For securing the communication between users in DeSCal, standard group communication security mechanisms can't be used despite the fact that users in DeSCal form shared calendar groups. It is described in Chapter 3 why this group communication security mechanisms are not appropriate for DeSCal.

5.1 Our proposed security framework

To meet all security requirements of DeSCal, we propose here a security framework complying with all the design requirements mentioned above. This security framework uses basic security protocols in such a way that it makes DeSCal secure and robust. It uses public key cryptography where each user is linked with a public key. Generally, to build a secure system of multiple users, it is very important to identify them in some way. Here, we identify a user by his public key. To associate a user (or user id) with a public key correctly, authentication of users' public keys is required and this first step is decisive for the correct working of rest of the security framework. As well, the authentication of users' public keys must be done in a decentralized way without using trusted third parties (TTPs). For this, a byzantine fault-tolerant public key authentication for pure Peer-to-Peer systems is proposed in [PI06] by Pathak and Iftode which suits best to our needs. For readability and completeness, we describe this protocol in brief below.

Pathak and Iftode's protocol

Pathak and Iftode [PI06] apply the ideas presented in the Byzantine Generals Problem [LSP82] for providing public key authentication in pure P2P systems, where generally one cannot assume the existence of a PKI. They postulate that a correct authentication depends on an honest majority of a particular subgroup of the peers' community, labeled "trusted group". However,

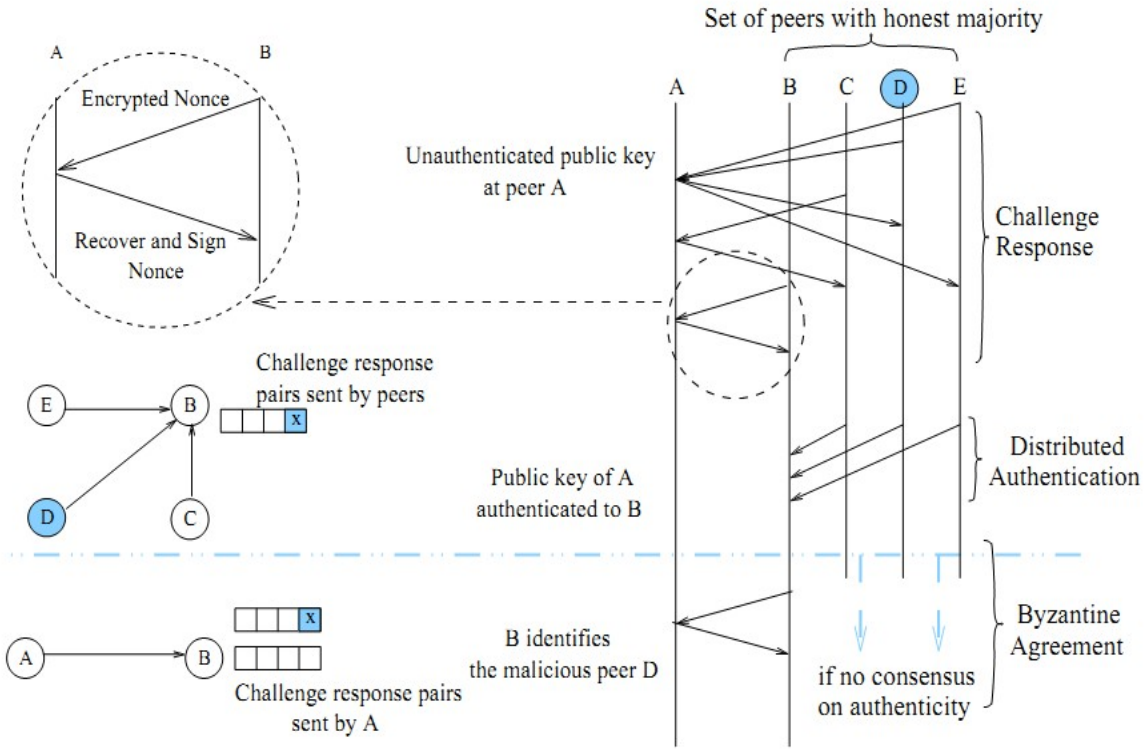

Figure 5.1: Authentication protocol example: A peer A is authenticated by B using its trusted peers. D is a malicious peer that tries to prevent authentication of A.

in this kind of systems, an authenticated peer could create multiple fake identities and could act maliciously in the future (Sybil attack [Dou02]). For this reason, the classification of the rest of the community maintained by each node has to be proactive and periodically flushed. Thus, honest members from trusted groups are used to provide a functionality similar to that of a CA (certification authority) through a consensus procedure.

The authentication protocol (see Figure 5.1, figure taken from the original paper [PI06]) consists of four phases: admission request, challenge response, distributed authentication and Byzantine agreement. The protocol begins when Bob runs into a newly discovered peer, Alice, with an unauthenticated public key (K_A), and then asks for the key to a subgroup of its trusted members, in order to verify its authenticity. Each notified peer challenges Alice by sending a random nonce encrypted with Alice's supposed public key (sent by Bob) in the signed challenge message. Alice will be able to return the recovered nonce in a signed response message (if and only if she holds the corresponding private key (K_A^{-1})). Each challenger waits for an application specific timeout, and if a correct response is received, he gets a proof of possession for K_A. All announced peers send their proofs of possession to Bob.

If all peers are honest, then there will be consensus and Bob will get the authentication result. Note that *Alice* or some of the peers can be detected as malicious or faulty if some votes differ. In this case, Bob first verifies if Alice is malicious by sending her the request message containing the proof. Alice must respond with all the challenge messages received and her respective

responses. If Alice can prove that she is not malicious, then some of the peers must be; in that situation, Bob must communicate a Byzantine fault to the group, which will send the Byzantine agreement message to others. All these transmitted messages have timestamps, source and destination identifiers, and digital signatures. Finally, successful authentication moves a peer to the trusted group, whereas encountered malicious peers are moved to the untrusted group.

For further details, we refer the reader to the original paper in [PI06].

Notations used

1. **Symmetric key**: A symmetric key used for event e is denoted as K_e.

2. **Public/Private key pair**: Public/private key pair of a user u is denoted as K_u and K_u^{-1} respectively.

3. **Symmetric cryptography**: Symmetric encryption and decryption of an event e using key K_e is denoted as E_{K_e}(e) and D_{K_e}(e) respectively.

4. **Asymmetric cryptography**: Asymmetric encryption and decryption of message 'm' using public/private key pair of user u is denoted as $\{m\}_{K_u}$ and $\{m\}_{K_u^{-1}}$ respectively.

5.1.1 Description

Here, we detail our proposed security framework and describe how it is enforced on top of coordination and access control model already employed by DeSCal [AIR11].

We present our security framework by describing the mechanism to be followed with respect to all possible happenings during the whole life cycle of DeSCal.

These possible happenings can be categorized based on their type:

1. **User-generated happenings**: This type of happenings are generated by a user by taking an action either on the shared calendar or on policy (by generating cooperative and administrative operations respectively).

2. **System-wide happenings**: A new user joins or an existing user leaves DeSCal. An existing user may disconnect for a while and then, may connect back *i.e.*, goes off-line and then, comes on-line again.

1. User-generated happenings

A user can take various cooperative and administrative operations at any point of time during DeSCal's lifetime. We describe the steps taken to enforce our security framework for each of these operations.

Inserting a new event. As all users in DeSCal are allowed to insert a new event. So, insertion of a new event is not checked against the local copy of the policy before it's execution at locally stored copy of the shared calendar. Once it's inserted in the local copy of the shared calendar, a cooperative request of insertion type is generated and broadcast to all other users in DeSCal. This cooperative request comprises the locally inserted new event which is to be inserted at all other copies of the shared calendar. Likewise, on reception of this cooperative request at other user sites, this new event is always allowed to be executed on remote copies of the shared calendar as all users are authorized to insert a new event in the shared calendar.

However, adding security framework requires the cooperative request to store the newly inserted event in some non-readable form which if broadcast, only authorized users can read. Extreme care has to be taken to construct this non-readable form keeping in mind all the security requirements of DeSCal and design requirements of security framework are satisfied.

When a user inserts a new event 'e' in the shared calendar, DeSCal needs to generate a new symmetric key 'K_e' corresponding to this newly created event. Next, the newly created event 'e' is encrypted by this symmetric key 'K_e' using a symmetric key encryption algorithm. Side-by-side, this symmetric key is first encrypted using the public key of the owner (who inserted this event 'e') and also, using public keys of other authorized users for 'Read' access to this event. All these encrypted forms constitutes a new form of event (which we call e', See 5.1). This new form e' can only be read by authorized users when broadcast in the group.

$$e' = E_{K_e}(e), \{K_e\}_{K_{Owner}}, \{K_e\}_{K_{AuthUser1}}, \{K_e\}_{K_{AuthUser2}}, ... \tag{5.1}$$

Just replacing the event *i.e.*, 'e' by 'e'', in insert operation of broadcast cooperative request helps us in keeping the coordination and access control models almost untouched while proposing this new security mechanism for insertion of an event. Also, it is very important to note here that it is e' which is stored in cooperative request, thereby, requiring no security mechanism for securing cooperative log. Also, confidentiality to the broadcast cooperative request is automatically provided during network transmission as no one (both inside or outside the group) except authorized users can read these events. We will describe later how other basic security properties are preserved for network communication.

This new form of event 'e'' consists of 1) encrypted event and 2) the symmetric key used for encryption of the event, encrypted separately with public key of the owner and all other authorized users. On reception of this broadcast cooperative request at remote user sites, only authorized users are to recover the event 'e' from 'e''. An authorized user can do this by getting the symmetric key used for the encryption of the event by decrypting the encrypted symmetric key with his private key and then, decrypting the event using this symmetric key. This ensures that only authorized users can get the event 'e' back but not other users.

Practically, it is possible that at the time of creation of an event, no other user is authorized to read this event and in this case, e' will take the following form (See 5.2):

$$e' = E_{K_e}(e), \{K_e\}_{K_{Owner}} \tag{5.2}$$

In 5.2, it can be easily seen that owner can always get the event 'e' back in case his site crashes; just by demanding the whole state of the shared calendar from any nearby user.

Deleting an existing event. When a user wants to delete an event in the shared calendar, it is checked against his local policy if he is authorized to do so or not. If authorized, this event is deleted from the local copy of the shared calendar and then, a cooperative request is generated which is to be broadcast to all other users in the group. On reception, this cooperative request is checked against the admin log and if authorized, the event in the local copy of the shared calendar is deleted. It must be noted that events are always stored in encrypted form 'e'' in the shared calendar and a view (the events for which he is authorized to read) is generated for the user from this encrypted calendar. In case of deletion, nothing extra has to be done with respect to our proposed security framework.

Editing an existing event. It is the job of the access control model already employed by DeSCal to ensure that an unauthorized user can't edit an existing event in the shared calendar. Edition of an event is same as inserting a new event from the perspective of our proposed security framework except the fact that editing an existing event doesn't always necessitate to generate a new symmetric key as is the case in inserting a new event. If the edition of the event by some authorized user is immediately after a revocation of 'Read' right to some user(s) by the owner, the editor must generate a new symmetric key because the set of authorized user(s) for 'Read' right has changed after this revocation operation. This new symmetric key is used for encrypting the edited event and the symmetric key is encrypted by the public key of the owner and new set of authorized user(s). It results into a new form of edited event e' where only authorized users can read the event in plain-text if it is broadcast to all other users in the group.

Also, we would like to point out that encryption of symmetric key with the public key of the owner and all authorized user is not needed to be computed and broadcast each time a user edits an event. Only if there is an immediate revocation of 'Read' right to a set of user(s) before the edition of an event, we need to generate this whole new form of event e' (See 5.1) corresponding to these newly set of authorized user(s) by generating a new symmetric key. Otherwise, we only need to compute and broadcast the encrypted edited event with the same symmetric key which is already being used for this event.

Granting 'Read' right. Only the owner of an event can grant a 'Read' right to a set of user(s). When the owner grants 'Read' right to a set of user(s), an administrative operation is generated after changing the locally stored policy. This administrative operation is broadcast to all other users in the group to make the change effective at policies stored at all other remote user sites. However, just broadcasting this administrative operation to grant 'Read' access to a set of user(s) on this event only enables them to change their locally stored policy. These 'Read' right granted user(s) still can't get access to the event in plain-text. For this, the owner must send the symmetric key used for this event encrypted with the public key of all granted user(s) along with this administrative operation. Suppose, the owner grants 'Read' access to two users u_1 and u_2 for some event 'e'. Symmetric key used for encrypting event 'e' is denoted by K_e. In this case, the owner must send information 'i' (See 5.6) along with the administrative operation.

$$i = \{K_e\}_{K_{u1}}, \{K_e\}_{K_{u2}} \tag{5.3}$$

By doing this, the users who are granted 'Read' access can first decrypt the symmetric key which is being used for encrypting this event. This can be done by decrypting their corresponding encrypted part of the symmetric key in the received information 'i' (along with the

administrative operation) using their private key. Once the symmetric key is obtained, one can use it to decrypt the encrypted event.

For the problem of how to broadcast this extra information 'i' with the administrative operation, we must look for a solution which minimizes changes in DeSCal's coordination and access control models. For the sake of this, we choose to send information 'i' embedded in the specified rule of this grant administrative operation. For each 'Read' right granted event, there must be an associated list which stores the symmetric key used for that event encrypted with the public key of all authorized user(s) in the same order as specified in that rule. It is easy to retrieve this information as each grant operation always has these keys encrypted with the 'Read' access granted users' public keys associated with the identifiers of the events in the rule. This requires us to make minor changes in the rule specified in the access control model already employed by DeSCal.

On reception of this information 'i' embedded in the rule, it is stored in the replicated encrypted shared calendar copy corresponding to the events in the rule. Once the event 'e' is retrieved from 'e'' at newly 'Read' access authorized users, DeSCal changes the view for these users by making this event available to read. We want to emphasize here on the fact that access control on shared calendar events in DeSCal is immediate and it strictly obeys the policy specified by the owner of the shared calendar events.

Revoking 'Read' right. In case of revocation, apart from sending the revocation administration request to all other users in the group, nothing has to be done by the owner from the perspective of our proposed security framework. Nevertheless, calendar view for these 'Read' access revoked users is changed to hide the events for which 'Read' access is revoked. All concurrent cooperative operations concerned by the revocation administrative operations are not allowed, thereby, maintaining high priority of administrative operations *i.e.*, once the 'Read' access to a set of events is revoked by the owner for a set of users, these users can no more read further updates to these events.

Later, if 'Read' access revoked event is modified by an authorized user, he constructs e' with respect to new set of authorized users. The set of authorized user has been changed by receiving the'Read' right revocation administrative operation. A new symmetric key has to be generated in this case (by the authorized user who modifies this event) and this event must be encrypted with this newly generated symmetric key so that these revoked 'Read' right users can no more access the event in plain-text using old symmetric key. A user can no more see the modifications of an event as soon as 'Read' right is revoked for a user.

2. System-wide happenings

Beyond any doubt, we need to consider all the possible happenings (apart from the user-generated happenings) in DeSCal while proposing this security framework for DeSCal. As a security framework is as strong as its weakest part, it must work correctly in all cases; which if not, may cause the system to halt and making it unavailable. Next, we look into the cases where users joins or leave shared calendar group in an arbitrary manner and then, we examine what happens when a user goes off-line and comes on-line again immediately. We illustrate how these cases are actually dealt with the proposed security framework.

A new user joins the shared calendar group. When a new user joins the shared calendar group after his public-key authentication, he contacts a nearby user with a request to join the shared calendar group. The contacted user sends his whole shared calendar state in persistent storage to this newly arrived user. As the whole shared calendar state in persistent storage is stored in encrypted form and no symmetric key corresponding to the shared calendar events is encrypted using the public key of this new user, he can't decrypt the encrypted events in the shared calendar state. Once the shared calendar state is retrieved by this newly arrived user, he can participate in the same manner as other existing users in the DeSCal.

An existing user leaves the shared calendar group. When a user leaves the shared calendar group, no extra care has to be taken with respect to the proposed security framework. Even DeSCal is silent to this happening and it's operation remains unaffected.

A user goes off-line and then, comes on-line again. When a user goes off-line, it is the same as a user leaves the shared calendar group. However, when a previous user come on-line at a later point of time, he retrieves the whole shared calendar state from a nearby user. From this retrieved encrypted shared calendar state, this user can get access to the events for which he is authorized (both his own events and the events of other users for which he is authorized to read).

How Fault-tolerance is achieved in DeSCal?

By design, DeSCal is fault-tolerant because it continues operating properly even if other user sites crashes or malfunctions. Also, our security framework is totally decentralized and is independent of other user sites' behavior, thereby, not losing its default feature by design. A user site running an instance of DeSCal has no effect if some other user sites crash or start behaving improperly.

Surviving a crash in DeSCal.

If a user site is crashed, the shared calendar state can be restored back. This property of DeSCal is unique and it is this property of DeSCal which makes it a favorite choice among users when they need to use either a shared or personal calendar application. After the crash at a user site, a user can again join the group asking the whole copy of the shared calendar. He can retrieve back all the events owned by him as well as the events for which he is authorized to 'Read'. The only information which a user must not lose is his private key and the whole shared calendar state can be restored back.

How availability of data is ensured?

We need to ensure that availability of data is not lost while adding this security framework on top of DeSCal. It is one among the most important characteristics of the DeSCal. If we lose availability of data, we lose the interest of full replication. However, at the cost of security, one may have to compromise a little bit with the availability because, to provide confidentiality to users' shared calendar events, these events must be stored in some non-readable form. In DeSCal, our need is to protect the confidentiality of the shared calendar events but at the same time, we also need to store this information at each user site. If we don't store the shared calendar events at unauthorized user sites, our other features of DeSCal like fault-tolerance, crash recovery and availability of data will be lost. To keep all these features preserved along

with providing confidentiality to users' calendar events, we need to store the shared calendar events in some non-readable form at unauthorized user sites. As seen above, fault-tolerance and crash recovery is achieved by DeSCal without any compromise and next, we show that availability of shared calendar events can also be ensured without any compromise.

In our proposed security framework, all shared calendar events are stored and broadcast in encrypted form such that a user can get access to the events in plain-text only if he is authorized to read them. As a user stores all the shared calendar events locally, the availability of data is ensured in the case when he can access them in plain-text. However, the shared calendar events where a user can't get access in plain-text, the issue of availability should arise. Such a case arises only when an owner of an event grants 'Read' right to some previously unauthorized user(s). When the owner grants 'Read' right to a user, this user must be able to access the event immediately. It is the case in our proposed security framework as the administrative operation granting 'Read' right for an event also contains the information which is needed to retrieve the event in plain-text. For all other cooperative and administrative operations, availability is already ensured as the whole shared calendar state is replicated at each user site.

5.1.2 Concurrency Issues

DeSCal already deals with all concurrency issues that arises from concurrent administrative and cooperative operations, though, new concurrency issues may arise with the addition of new 'Read' right administrative operation in our proposed security framework. In this section, first, we investigate the operations where concurrency issues may arise with respect to 'Read' right administrative operation and then, we justify them with a slight modification of coordination and access control models already employed by DeSCal.

First of all, it is obvious that 'Read' right administrative operation for an event 'e' can't be concurrent to administrative operations of other types for this event. This is because administrative operations are generated by a single user (*i.e.*, only by the owner of an event) and always ordered.

Also, 'Read' right administrative operation can only be specified after creating/inserting an event by the owner and can't be concurrent to cooperative operation of insertion type. Only an owner can insert an event in the shared calendar and generate 'Read' right administrative operation on the events administered by him. This implies that there can't be a case where two different users concurrently generate a cooperative operation of insertion type and 'Read' right administrative operation for the same event.

Cooperative operation of deletion type and 'Read' right administrative operation for the same event can be concurrent as an authorized user can delete the event whereas at the same time, administrator of that event can generate a 'Read' right administrative operation. However, below, we investigate if this concurrency of two operations can cause some problems or not. A cooperative operation of deletion type for an event 'e' corresponds to simply deleting this event from the shared calendar and it doesn't make any sense if an event is readable or not at a particular user site at the time of deletion of this event. In actual, an event to be deleted is identified by the position of that event in the shared calendar (and not by the event itself) and this leads to having no concurrency issues in case of these two concurrent operations.

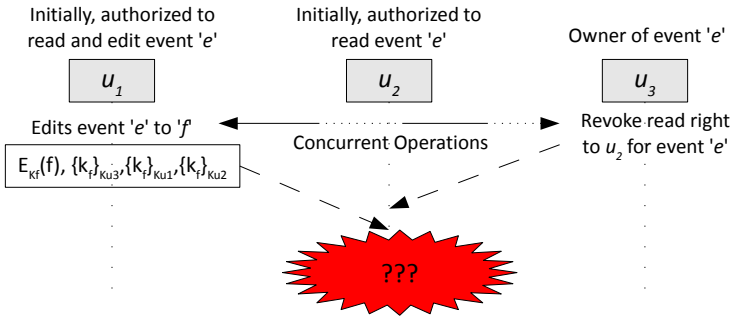

Figure 5.2: Necessity of undo in case of concurrent right revocation and edit operations

Finally, as an event may be edited by an authorized user at the same time when the owner of that event generates a 'Read' right administrative operation, it is evident that concurrent cooperative operation of update type and 'Read' right administrative operation can be generated. There are concurrency issues in this case which bothers correct working of DeSCal and next, we investigate these issues further for 'Read' right grant and revocation administrative operations separately.

'Read' right revocation and 'Edit' concurrent operations.

Suppose there are three users: u_1, u_2 and u_3. An authorized user u_1 edits an event while the user u_3 (administrator) of that event revokes 'Read' right for user u_2 (See Figure **??**). As user u_1 is unknown of the concurrent revocation of 'Read' right to user u_2, he generates the new form f' of edited event f as shown in 5.4.

$$f' = E_{K_f}(f), \{K_f\}_{K_{f_3}}, \{K_f\}_{K_{f_1}}, \{K_f\}_{K_{u_2}} \tag{5.4}$$

The generated cooperative operation corresponding to this edit operation consists of this new form f' and is broadcast to other users after editing the event in locally stored shared calendar at user site u_1. As f' contains the symmetric key used to encrypt this edited event f, this implies that edited event f by user u_1 can be read by u_2, however, it should not be the case as in DeSCal, administrative operations are given higher priority than cooperative operations in case two cooperative and administrative operations are concurrent.

The access control model already employed by DeSCal is based on an 'UNDO' mechanism in case of concurrent restrictive administrative operations and the solution to this above problem also consists in undoing the edit operation by user u_1, thereby, enforcing administrator's 'Read' right revocation operation. For the sake of reader's information, the access control model employed by DeSCal is inspired from [ICR09], a flexible access control model for distributed collaborative editors. We refer the reader to [ICR09] to better understand this 'UNDO' mechanism.

'Read' right grant and 'Edit' concurrent operations.

In case of 'Read' right grant administrative operation, the administrator of an event sends the administrative request which also contains the information using which an authorized user can decrypt back the encrypted event. It seems like perfectly okay at first glance in case of 'Read' right grant administrative operation but, we present below a case where the problem can occur.

The problem can occur if a user edits an event[5] and the administrator of this event concurrently attributes 'Read' right to a set of user(s) (See Figure 5.3). In this figure, user u_4 (owner of event 'e') revokes read right to u_2 for the event 'e' who was previously authorized to read this event. Next, user u_4 attributes read right for event 'e' to user u_3 and concurrently, user u_1 (who is authorized to edit the event 'e') edits event 'e'. User u_1 generates a new symmetric key as there was an immediate revocation for this event before this edition and encrypts this edited event 'f' with this new key. Also, he encrypts the symmetric key used for encrypting the event with the public key of the owner u_4 and other users who are authorized to read this event (in this case, no other users apart from him (u_4) is authorized). It results into a new form f' ((See 5.5)) of this edited event f.

$$f' = E_{K_f}(f), \{K_f\}_{K_{u_4}}, \{K_f\}_{K_{u_1}} \tag{5.5}$$

Later, user u_1 broadcasts the generated cooperative request containing this new form f' of edited event f. At user site u_4, the concurrent 'Read' right grant administrative operation modifies the local copy of the policy and then, generated administrative request is broadcast. This administrative request also contains the information 'i' (See 5.6) so that newly 'Read' right granted user u_3 can decrypt the encrypted event, by first getting the symmetric key used for encrypting the event and then, using this symmetric key to decrypt the event.

$$i = \{K_e\}_{K_{u_3}} \tag{5.6}$$

Suppose the cooperative request generated by user u_1 reaches at user site u_3 first and later, u_3 receives the 'Read' right grant administrative request. When user u_3 receives the 'Read' right grant administrative request, he will try to use the received key K_e to decrypt the encrypted event, but he won't be able to able to do so as the stored encrypted form is modified. Again, the solution to this problem is to undo the concurrent edit operations with respect to 'Read' right grant administrative operations. In general, a cooperative request of update type is undone in case of concurrent 'Read' right administrative operation.

5.2 An illustrating example

To better understand our proposed security framework for 'Read' access control on shared calendar events, we present a scenario (See Figure 5.4) comprising of three users: u_1, u_2 and u_3. Each user simultaneously inserts two events in the shared calendar and it is the responsibility of the coordination module to maintain the consistency of the shared calendar. After all, each user will have the same converged, consistent copy of the shared calendar. Here, in our setting, events e_1 and e_5 are created by user u_1; e_3 and e_6 are created by u_2 whereas e_2 and e_4 are created by u_3. User u_1 authorizes u_2 and u_3 to read his events e_1 and e_5 respectively; u_2 authorizes u_1 and u_3 to read his events e_3 and e_6 respectively whereas u_3 authorizes u_1 and u_2 to read his events e_2 and e_4 respectively. Here, we highlight that each user has the same state of the shared calendar but a user is only able to read the events for which he is authorized. For example, at each user site, event e_1 is stored in some encrypted form (See 5.7)

$$Encrypted\ form\ of\ e_1 = E_{K_{e_1}}(e_1),\ \{K_{e_1}\}_{K_{u_1}},\ \{K_{e_1}\}_{K_{u_2}} \tag{5.7}$$

[5]With the assumption that this edit is followed by a revocation; thereby, necessitating a change of symmetric key to encrypt the edited event.

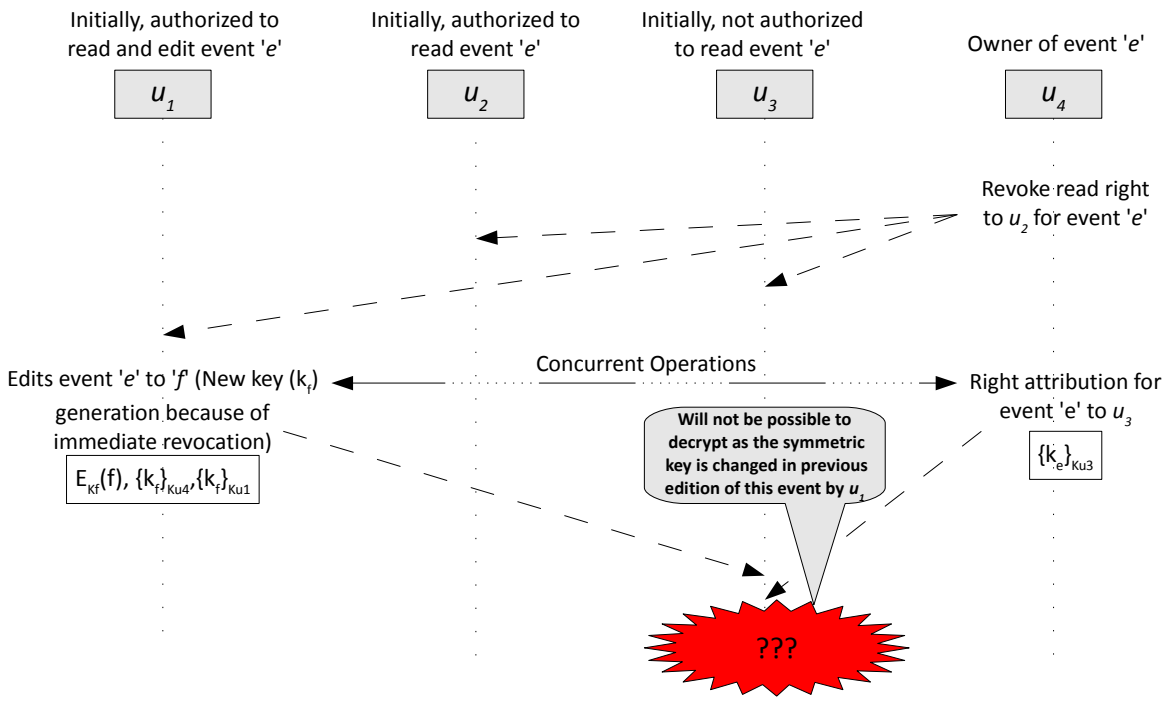

Figure 5.3: Necessity of undo in case of concurrent right attribution and edit operations

In this encrypted form, along with encrypted event ($E_{K_{e_1}}(e_1)$) using a symmetric key (K_{e_1}), this symmetric key used to encrypt the event e_1 is encrypted using the public keys K_{u_1} and K_{u_2} of users u_1 and u_2 respectively. This implies that only owner u_1 and authorized user u_2 can read this event e_1 stored in this encrypted form at each user site. Moreover, this event e_1 is stored at each user site in such a manner that all three salient features of DeSCal (fault-tolerance, availability and crash recovery) are satisfied. It is easily evident that event e_1 can be retrieved back by user u_1 and u_1 by asking the copy of it from u_3.

New user u_4 joins the group.

When a new user u_4 joins the group and contacts a nearby user u_2 to get the whole shared calendar state. Irrespective of having the whole shared calendar state, this new user u_4 can't retrieve any event in plain-text as he is not authorized to read them. After getting the whole shared calendar state, this new user u_4 can take part in the shared calendar in the same manner as other existing users u_1, u_2 and u_3.

User site u_1 is crashed.

If user site u_1 is crashed, u_1 will not lose his shared calendar events as long as he has his private key somewhere safe. Moreover, u_1 will not only retrieve his own shared calendar events but also, other users' events for which he is authorized to read; effectively, restoring the whole shared calendar state back. In this case, user u_1 will use his private key to decrypt the symmetric keys corresponding to the events for which he is the owner or he is authorized to read. Even a user can use DeSCal as his personal calendar by not sharing his events with other users in the group and can take the benefit of crash recovery.

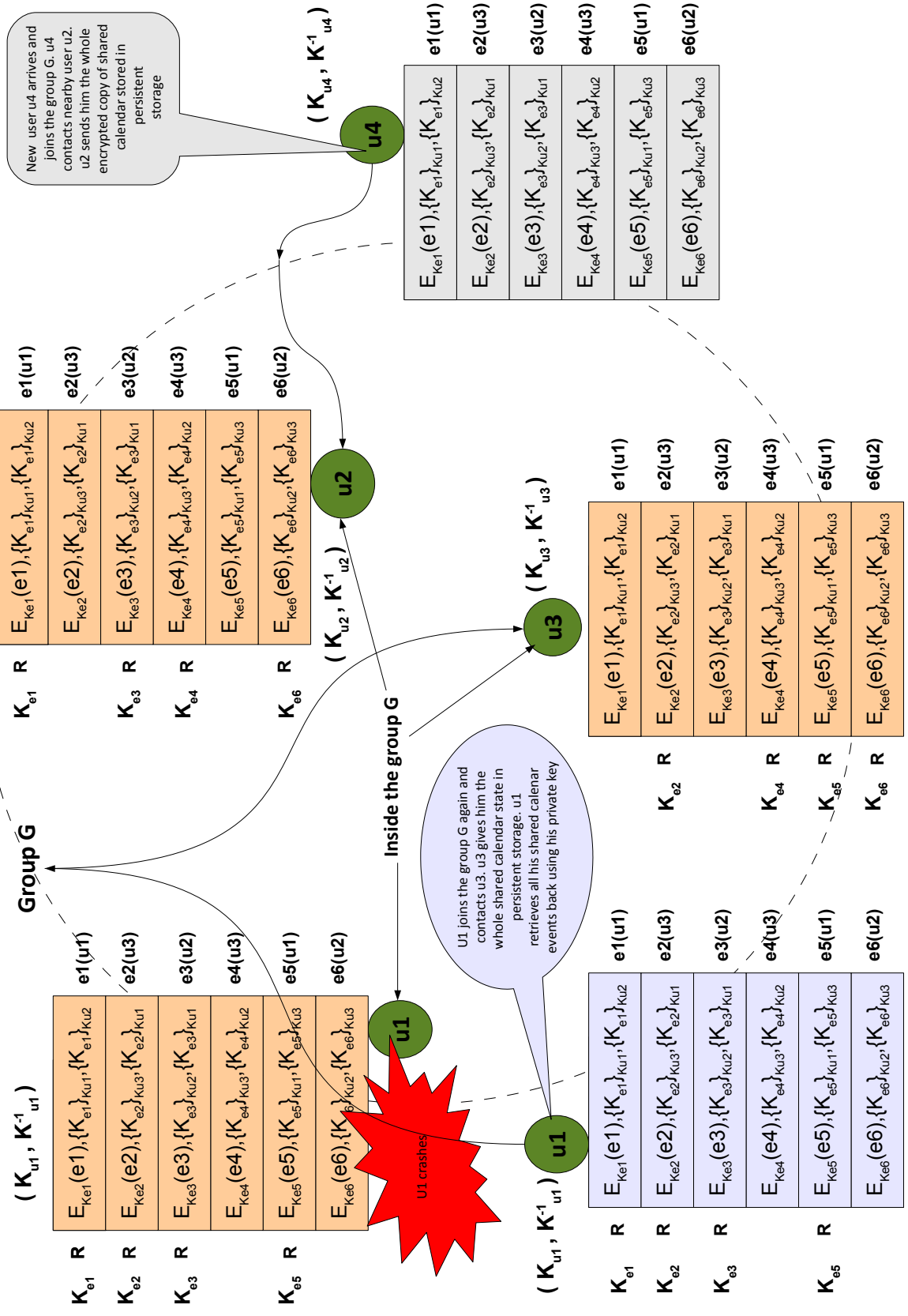

Figure 5.4: Example illustrating our proposed security framework

5.3 Securing the communication between users

We would like to point out that securing the communication between users was also left as future work in the original paper of DeSCal [AIR11]. Here, we highlight that our proposed security framework for 'Read' access control automatically takes care of confidentiality of the shared calendar events while they are in transit. This is due to the fact that events are always sent in encrypted form where the symmetric key used for encrypting the event is encrypted only with the public key of the users who are authorized to read this event, thereby, allowing only authorized users (inside or outside the group) to decrypt back the encrypted event (by decrypting the encrypted symmetric key using their private key). No one else apart from authorized users can read the encrypted event by controlling the communication channel between users of DeSCal.

To avoid replay attack, a network message is associated with a number (randomly chosen for the first message and then, incrementing it with each network message sent). The receiver can detect a replay message if the counter value is repeated. To ensure integrity, authenticity and non-repudiation, the network message is first hashed[6] to produce a short digest that is then signed (using the private key of the sender).

For illustration, we denote a message (*e.g.*, cooperative or administrative request or state of the calendar) by 'm' which is to be broadcast by user u_i. To avoid replay attack, we propose to associate a randomly generated *counter* with message 'm' and we denote it by 'm'' (See 5.8).

$$m' = \{m,\ counter\} \tag{5.8}$$

Next, to guarantee integrity, authenticity and non repudiation of message 'm'', it is first hashed to produce a short digest that is then signed with private key ($K_{u_i}^{-1}$) of user u_i. It is denoted by *sig* (See 5.9). In the end, m'' (See 5.9) is broadcast to other users in the shared calendar group.

$$m'' = \{m', sig\}\ where\ sig = \{hash(m')\}_{K_{u_i}^{-1}} \tag{5.9}$$

5.4 Discussion

In this proposed security framework, a new symmetric key has to be generated by the owner for each new event created in the shared calendar. It increases the computational cost while inserting a new event. Also, the symmetric key used for encrypting an event has to be stored encrypted with public key of the owner and all authorized users at each user site. It leads to extra storage requirements. However, our proposed solution is not expensive as the data to be encrypted (shared calendar events) is normally not huge. Moreover, this solution preserves all characteristic features of DeSCal (*i.e.* fault-tolerance, crash recovery, availability) while providing security to DeSCal at the same time. We believe that one has to compromise a little bit of computational cost and storage wastage at the cost of preserving characteristic features of DeSCal.

[6] A cryptographic hash function is a deterministic procedure that takes an arbitrary block of data and returns a fixed-size bit string, the (cryptographic) hash value, such that an intentional or accidental change to the data will change the hash value.

Chapter 6

Implementation on iPhone OS

Our proposed security framework has been implemented on top of iPhone OS implementation of DeSCal system. For details on implementation of DeSCal on iphone OS, please, refer to [AIR11].

At the time of writing this report, byzantine fault-tolerant public key authentication [PI06] by Pathak and Iftode is not implemented. However, we plan to implement it in near future and one can ensure that it's implementable as the authors of [PI06] has already implemented this protocol as a standalone library to make it available to a variety of applications. In the current implementation of the security framework, a user sends his public key to all other users in DeSCal when he joins the group. Also, when a new user joins the group, existing users in the group send this newly arrived user their public key. This will lead to a user in DeSCal possessing the public key of all other users. As a user possesses the public key of all other users in DeSCal which is actually necessary for our proposed security framework to work, it starts operating as explained in Chapter 5.

The current implementation of our security framework on iPhone OS uses RSA algorithm for asymmetric encryption and public/private key pair of size 1024 bits. For symmetric encryption, it uses AES-128.

Figure 6.1 contains four snapshots of our iPhone OS implementation of DeSCal (Calendar, Event Detail, Policy and Available Peers view from left to right).

For demonstration, let's take an example where a user wants to give '*Read*' right to a user named 'Michael' for his events: <Event 1> and <Event 2> where <Event 1> = {Title:Concert Location:Stanislas Date & Time:24 June 2011 09:00:04 PM} and <Event 2> = {Title:See Tigran Location:A201 Date & Time:15 July 2011 10:25:04 AM}. So, in this case, we will have to select 'Michael' from the list of users, 'Event 1' and 'Event 2' from the list of events, '*Read*' from the list of rights (*Read, Delete* & *Edit*) and '*Right Attribution*' from the list of permissions(*Right Attribution* & *Right Revocation*). See Figure 6.2 to have a look on how this rule selection is performed in iPhone OS implementation of DeSCal.

Figure 6.1: Calendar, Event Detail, Policy and Available Peers view in iPhone OS implementation of DeSCal

Figure 6.2: Selection of various attributes to insert a new rule in policy in iPhone OS implementation

Conclusions and Possible Directions of Future Work

1 Conclusions

Securing DeSCal had been mentioned as future work in the original paper of DeSCal [AIR11] and we dealt with it here. We have designed and implemented a security framework for DeSCal which ensures confidentiality and integrity of shared calendar events. Also, the network communication between users is secured with respect to other basic security properties like non-repudiation, message authentication, replay attack. The proposed security framework uses byzantine fault tolerant public key authentication in pure Peer-to-Peer systems [PI06] by Pathak and Iftode for users' public-key authentication. It makes use of both symmetric and asymmetric encryption in such a way that all security requirements of DeSCal are satisfied.

2 Possible Directions of Future Work

First of all, we plan to formally analyze and verify our proposed security framework using tools like AVISPA [ABB+05] which automatically perform the verification of security protocols. Inspired form [Cal], another dimension of formalizing our work is to specify the communication protocol in a decentralized manner as specified in [Cal] for centralized shared calendars.

Also, DeSCal requires a user-authentication mechanism where users are allowed to join the shared calendar group based on some predefined policy. Currently, DeSCal allows everyone to join in an Ad-Hoc manner and participate in the shared calendar. However, this may cause a problem for other users in DeSCal if a malicious user joins the shared calendar group and starts inserting useless events just to overload the events in the shared calendar. We also plan to fix this problem in future.

Apart from this, we plan to look for more efficient solution which can satisfy the security requirements of DeSCal while preserving its characteristic features. In this direction, we plan to investigate few works described below and ascertain if these works can be adapted to meet security demands of DeSCal.

ID-based encryption (or Identity-Based Encryption (IBE)) is an important primitive of ID-based cryptography. ID-based encryption was proposed by Adi Shamir in 1984. As such it is a type of public-key encryption in which the public key of a user is some unique information about the identity of the user (e.g. a user's email address). In 2004, Sahai and Waters introduce a new type of IBE scheme in their paper [SW04] that they call Fuzzy IBE. In Fuzzy IBE they view an

identity as a set of descriptive attributes. A Fuzzy IBE scheme allows for a private key for an identity, ω, to decrypt a cipher-text encrypted with an identity, ω', if and only if the identities ω and ω' are close to each other as measured by the matching of their descriptive attributes on a specific type of distance metric. The motivation behind developing Fuzzy IBE was to utilize its error-tolerance property to enable encryption using biometric inputs as identities, which inherently have some noise each time they are sampled. Later, a new cryptosystem for fine-grained sharing of encrypted data is developed in [GPSW06] where cipher-texts are labeled with sets of attributes and private keys are associated with access structures that control which cipher-text a user is able to decrypt. The Attribute-Based Encryption systems used attributes to describe the encrypted data and built policies into user's keys; while in [BSW07], attributes are used to describe a user's credentials, and a party encrypting data determines a policy for who can decrypt. [BSW07] is conceptually similar to Role-Based Access Control (RBAC). As [BSW07] presents a system for realizing complex access control on encrypted data and it is conceptually similar to RBAC, we plan to investigate further this work if it can be used more efficiently to satisfy the security requirements of DeSCal.

Also, Broadcast encryption [FN94] is the cryptographic problem of encrypting broadcast content (e.g. TV programs) in such a way that only qualified users (e.g. subscribers who've paid their fees) can decrypt the content. In broadcast encryption, a central broadcast site can broadcast secure transmissions to an arbitrary set of recipients while minimizing key management related transmissions. This paper presents several schemes that allow a center to broadcast a secret to any subset of privileged users out of a universe of size n so that coalitions of k users not in the privileged set cannot learn the secret. The most interesting scheme requires every user to store O(k logk logn) keys and the center to broadcast O(k^2 $\log^2$klog(1/p)) messages regardless of the size of the privileged set. Several solutions exist offering various trade-offs between the increase in the size of the broadcast, the number of keys that each user needs to store, and the feasibility of an unqualified user or a collusion of unqualified users being able to decrypt the content. Our future work consists of analyzing the technique of broadcast encryption if it can be suitable to the security needs of DeSCal and if yes, make a comparison between it and our proposed security framework. This comparison will be done based on various factors like performance, key management techniques, storage cost, message broadcast size.

Bibliography

[AAEM09] Divyakant Agrawal, Amr El Abbadi, Fatih Emekci, and Ahmed Metwally. Database management as a service: Challenges and opportunities. *Data Engineering, International Conference on*, 0:1709–1716, 2009.

[ABB+05] Alessandro Armando, David Basin, Yohan Boichut, Yannick Chevalier, Luca Compagna, Jorge Cuellar, Paul Hankes Drielsma, Pierre-Cyrille Heám, Jacopo Mantovani, Sebastian Mödersheim, David von Oheimb, Michaël Rusinowitch, Judson Santiago, Mathieu Turuani, Luca Viganò, and Laurent Vigneron. The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications. In Kousha Etessami and Sriram K. Rajamani, editors, *Proceedings of the 17th International Conference on Computer Aided Verification (CAV'05)*, volume 3576 of *LNCS*. Springer, 2005. Available at `http://www.avispa-project.org/publications.html`.

[AD06] W.J. Adams and N.J. Davis. TMS: a trust management system for access control in dynamic collaborative environments. In *Performance, Computing, and Communications Conference, 2006. IPCCC 2006. 25th IEEE International*, pages 8 pp. –150, april 2006.

[Ada06] William Joseph Adams. *Decentralized trust-based access control for dynamic collaborative environments*. PhD thesis, Blacksburg, VA, USA, 2006.

[AIR11] Jagdish Prasad Achara, Abdessamad Imine, and Michael Rusinowitch. DeSCal—Decentralized Shared Calendar for P2P and Ad-Hoc Networks. To appear in the 10th International Symposuim on Parallel and Distributed Computing (ISPDC'2011), Cluj-Napuca, Romania, July 6th - 8th, 2011.

[AIS+05] William Joseph Adams, Dr. Nathaniel J. Davis Iv, Dr. Scott, F. Midkiff, Dr. William, T. Baumann, Dr. Edward, L. Green, and William J. Adams. Toward a decentralized trust-based access control system for dynamic collaboration. In *IEEE Workshop on Information Assurance*, page 324, 2005.

[BCSV94] Carlo Blundo, Antonella Cresti, Alfredo De Santis, and Ugo Vaccaro. Fully dynamic secret sharing schemes. In *Theoretical Computer Science*, pages 110–125. Springer-Verlag, 1994.

[Bla79] G. R. Blakley. Safeguarding cryptographic keys. *Managing Requirements Knowledge, International Workshop on*, 0:313, 1979.

[BSW07] John Bethencourt, Amit Sahai, and Brent Waters. Ciphertext-policy attribute-based encryption. In *Proceedings of the 2007 IEEE Symposium on Security and*

Privacy, SP '07, pages 321–334, Washington, DC, USA, 2007. IEEE Computer Society.

[Cac95] Christian Cachin. On-line secret sharing. In *In Proc. of the 5th IMA Conf. on Cryptography and Coding*, pages 90–198. Springer-Verlag, 1995.

[Cal] CalDAV. `http://caldav.calconnect.org/`.

[Dou02] John R. Douceur. The sybil attack. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 251–260, London, UK, 2002. Springer-Verlag.

[FN94] Amos Fiat and Moni Naor. Broadcast encryption. In *Proceedings of the 13th annual international cryptology conference on Advances in cryptology*, pages 480–491, New York, NY, USA, 1994. Springer-Verlag New York, Inc.

[GCa] Google Calendar. `http://www.google.com/googlecalendar/about.html`.

[GJO+06] Elizabeth Gray, Christian Jensen, Paul O'Connell, Stefan Weber, Jean-Marc Seigneur, and Yong Chen. Trust evolution policies for security in collaborative ad hoc applications. *Electronic Notes in Theoretical Computer Science*, 157(3):95 – 111, 2006. Proceedings of the First International Workshop on Security and Trust Management (STM 2005).

[GKLL] Roxana Geambasu, Tadayoshi Kohno, Amit A. Levy, and Henry M. Levy. Vanish: Increasing data privacy with self-destructing data.

[GPSW06] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *Proceedings of the 13th ACM conference on Computer and communications security*, CCS '06, pages 89–98, New York, NY, USA, 2006. ACM.

[Her86] Maurice Herlihy. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems*, 4:32–53, 1986.

[HT88] Maurice Herlihy and J. D. Tygar. How to make replicated data secure. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, CRYPTO '87, pages 379–391, London, UK, 1988. Springer-Verlag.

[ICR09] Abdessamad Imine, Asma Cherif, and Michaël Rusinowitch. A flexible access control model for distributed collaborative editors. In *Proceedings of the 6th VLDB Workshop on Secure Data Management*, SDM '09, pages 89–106, Berlin, Heidelberg, 2009. Springer-Verlag.

[Imi09] Abdessamad Imine. Coordination model for real-time collaborative editors. In *Proceedings of the 11th International Conference on Coordination Models and Languages*, COORDINATION '09, pages 225–246, Berlin, Heidelberg, 2009. Springer-Verlag.

[JAV08] Mohamed Jawad, Patricia Serrano Alvarado, and Patrick Valduriez. Design of priserv, a privacy service for dhts. In *Proceedings of the 2008 international workshop on Privacy and anonymity in information society*, PAIS '08, pages 21–25, New York, NY, USA, 2008. ACM.

[Kra94] Hugo Krawczyk. Secret sharing made short. In *Proceedings of the 13th annual international cryptology conference on Advances in cryptology*, pages 136–146, New York, NY, USA, 1994. Springer-Verlag New York, Inc.

[Liu08] Yichun Liu. Trust-Based Access Control for Collaborative System. *Computing, Communication, Control and Management, ISECS International Colloquium on*, 1:444–448, 2008.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4:382–401, July 1982.

[LWR09] Min Li, Hua Wang, and D. Ross. Trust-Based Access Control for Privacy Protection in Collaborative Environment. In *e-Business Engineering, 2009. ICEBE '09. IEEE International Conference on*, pages 425 –430, oct. 2009.

[MMT01] Ayako Maeda, Atsuko Miyaji, and Mitsuru Tada. Efficient and unconditionally secure verifiable threshold changeable scheme. In *In ACISP 2001, volume 2119 of LNCS*, pages 402–416. Springer - Verlag, 2001.

[PI06] Vivek Pathak and Liviu Iftode. Byzantine fault tolerant public key authentication in peer-to-peer systems. *Comput. Netw.*, 50:579–596, March 2006.

[PTHCR08] Esther Palomar, Juan M. E. Tapiador, Julio C. Hernandez-Castro, and Arturo Ribagorda. Secure content access and replication in pure p2p networks. *Comput. Commun.*, 31:266–279, February 2008.

[Rab89] Michael Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36:335–348, 1989.

[Riv97] Ronald L. Rivest. All-or-nothing encryption and the package transform. In *In Fast Software Encryption, LNCS*, pages 210–218. Springer-Verlag, 1997.

[RP11] J. K. Resch and J. S. Plank. AONT-RS: blending security and performance in dispersed storage systems. In *FAST-2011: 9th Usenix Conference on File and Storage Technologies*, February 2011.

[RRT+09] Venugopalan Ramasubramanian, Thomas L. Rodeheffer, Douglas B. Terry, Meg Walraed-sullivan, Ted Wobber, Catherine C. Marshall, and Amin Vahdat. Cimbiosys: A platform for content-based partial replication. nsdi, 2009.

[SB05] Arun Subbiah and Douglas M. Blough. An approach for fault tolerant and secure data storage in collaborative work environments. In *Proceedings of the 2005 ACM workshop on Storage security and survivability*, StorageSS '05, pages 84–93, New York, NY, USA, 2005. ACM.

[SG07] Mark W. Storer and Kevin M. Greenan. Potshards: secure long-term storage without encryption. In *In Proceedings of the 2007 USENIX Annual Technical Conference*, pages 143–156, 2007.

[Sha79] Adi Shamir. How to share a secret. *Commun. ACM*, 22:612–613, November 1979.

[SW04] Amit Sahai and Brent R. Waters. Fuzzy identity based encryption, 2004.

[SWP04] Ron Steinfeld, Huaxiong Wang, and Josef Pieprzyk. Lattice-based threshold-changeability for standard shamir secret-sharing schemes. In *In Asiacrypt'04, volume 3329 of LNCS*, pages 170–186. Springer - Verlag, 2004.

[TW] Christophe Tartary and Huaxiong Wang. Dynamic threshold and cheater resistance for shamir secret sharing scheme.

[WRT09] Ted Wobber, Thomas L. Rodeheffer, and Douglas B. Terry. Policy-based access control for weakly consistent replication, 2009.

[WXYX07] Chen Wei, Long Xiang, Bai Yuebin, and Gao Xiaopeng. A new dynamic threshold secret sharing scheme from bilinear maps. *Parallel Processing Workshops, International Conference on*, 0:19, 2007.

[Zim] Zimbra Platform Calender Application. `http://www.zimbra.com/products/calender-collaboration.html`.

Résumé

Nous proposons un protocole de sécurité pour des agendas partagés dont la gestion de données est complètement décentralisée. Dans ce protocole, nous assurons à la fois (i) la confidentialité du contenu répliqué et (ii) la sécurité de communication entre les utilisateurs. Comme nous utilisons une réplication complète de données, notre protocole préserve toutes les caractéristiques d'une telle réplication, à savoir : la tolérance aux pannes et la reprise après panne. Pour valider notre solution, nous avons implémenté un prototype sur des mobiles tournant sous le système iPhone OS.

Mots-clés: Sécurité, Réplication, Confidentialité, Intégrité, Décentralisation, Disponibilité, Tolérance aux pannes, Reprise sur incident

Abstract

We propose a security framework for Decentralized Shared Calendar. The proposed security framework provides confidentiality to replicated shared calendar events and secures the communication between users. It is designed in such a way that DeSCal preserves all of its characteristic features like fault-tolerance, crash recovery, availability and dynamic access control. It has been implemented on iPhone OS.

Keywords: Security, Replication, Confidentiality, Integrity, Decentralization, Availability, Fault-tolerance, Crash Recovery