A CLOUD STORAGE PLATFORM IN THE DEFENSE CONTEXT
Mobile Data Management With Unreliable Network Conditions

Jan Sipke van der Veen[1], Mark Bastiaans[2], Marc de Jonge[1] and Rudolf Strijkers[2,3]

[1]*TNO, Groningen, The Netherlands*

[2]*TNO, Delft, The Netherlands*

[3]*University of Amsterdam, Amsterdam, The Netherlands*

{*jan_sipke.vanderveen, mark.bastiaans, marc.dejonge*}*@tno.nl, strijkers@uva.nl*

Keywords: Cloud Storage, Storage Platform, Mobile Data, Synchronization, Caching, Discovery.

Abstract: This paper discusses a cloud storage platform in the defense context. The mobile and dismounted domains of defense organizations typically use devices that are light in storage, processing and communication capabilities. This means that it is difficult to store a lot of information on these devices locally, but also that it is infeasible to rely on a central storage system that is accessible through a network. The concept of Information of Interest (IoI) is introduced to denote the information demand of a user and its devices and applications. A novel storage platform is designed and tested that uses well-known techniques such as synchronization, caching and discovery, and uses the IoI to determine the storage strategy. A sample application was created that runs on personal computers, mobile phones and tablets. Manual and automated tests were run to show that the platform behaves as expected.

1 INTRODUCTION

The last decades have shown the importance of information superiority in the defense context (Thomas, 2000). The information used in this context ranges from high level information at the top of the defense organization all the way down to the detailed information of the individual soldier.

The static and deployed domains of defense organizations typically use devices with high storage and processing capacity, which are connected with a reliable, high bandwidth network. This means that information can be stored on the devices themselves or in a central storage system and accessed through the network when needed.

However, the mobile and dismounted domains of a defense organization typically use devices that are light, not only in weight but also in storage, processing capacity and communication capabilities. This limits the amount of information that the device can store locally. To make matters worse, the soldier is regularly faced with unreliable and low bandwidth networks, making remote information access difficult as well (Burbank et al., 2006). Standard cloud storage platforms - both public such as S3 (Amazon, 2011) and private such as Swift (OpenStack, 2011) - depend on a reliable network connection with their users, which is why they cannot be used as-is in mobile and dismounted domains.

Another problem defense organizations face is that of unintended interaction between applications. Each application typically uses the storage capacity of the device it runs on and, when needed, uses the network to communicate with a server elsewhere. This may result in applications interfering with each other. For example, an important application may need extra storage capacity or network bandwidth, while a less important application is using this capacity. Without a layer between the applications and the resources they use, one application cannot have precedence over others.

There are standard techniques for caching information on a local device (Halinger and Hohlfeld, 2010) (Nguyen and Dong, 2011) and accessing information remotely when needed, but they typically assume a reliable, high bandwidth network connection between devices (Bohossian et al., 2001) (Rhea et al., 2001). Other techniques try to deal with slow networks (Suel et al., 2004) (Shinkuma et al., 2011), but do not use the domain knowledge of applications that use this storage.

This paper proposes a novel platform that combines a set of well-known techniques that does not depend on a reliable, high bandwidth network. It can continue storing and retrieving information locally, until the network is available again. At the same time, it can prioritize applications that need more capacity at the expense of less important applications. It also makes use of the domain knowledge of the applications to decide which pieces of information may be deleted when storage capacity runs out.

2 INFORMATION OF INTEREST

It is both impractical and unnecessary to present soldiers in the field with all information the army has its disposal. The soldier's devices simply contain too little storage capacity to hold all information. And even if the devices could hold so much information, the soldier would be overwhelmed by it (Hancock and Szalma, 2008).

A selection therefore needs to be made of the total information space that is available. Good examples of selection criteria for a soldier are: the author of the information (e.g. the commanding officer), the location that the information refers to (e.g. a town nearby) and the time when the information was entered. The term Information of Interest (IoI) formalizes this as a technical representation of information demand of a user and its devices and applications.

Figure 1 contains an example of users A, B and C with their own IoI and some shared interests. These shared interests depend on the type of information, e.g. B and C have an overlap in terrain information interest, but none in information about persons. This makes it possible for information to be shared among users, but only the information that they are all interested in.

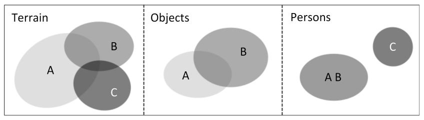


Figure 1: Information of Interest of parties A, B and C with overlapping interests, depending on type of information.

3 STORAGE PLATFORM

Given the defense context there are several possible requirements on a storage solution, but the one we focus on in this paper is the ability to store all and retrieve the most important information when there is no or bad network connectivity.

The CAP theorem (Brewer, 2000) states that you can obtain at most two out of three properties in a shared data system: consistency (C), availability (A) and tolerance to network partitions (P). As we can see from our requirement, tolerance to network partitioning is important, as well as availability. This means that consistency is no longer attainable and we should deal with this by resolving conflicts that might arise.

Figure 2 shows how our platform resides in between the local storage and the applications. Synchronization with other devices takes place through the network. Queries for information (e.g. get items with a certain author and within a specified time range) and commands to store information (e.g. create a new item or update the contents of an existing one) all pass through this layer. Also, an interface to applications is provided to set the IoI and resolve conflicts that cannot be handled automatically.

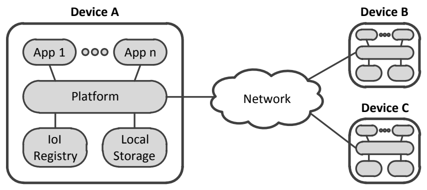


Figure 2: Platform on each device.

Because of the potentially bad network connectivity between the devices, there is no central entity the platform can depend on. Instead, devices find one another by means of a distributed discovery mechanism (see section 3.1).

In a distributed system with unreliable network connections, data needs to be stored locally when an application creates or updates an information item. If the network is available at that time or becomes available again later, this item can be synchronized with partners (see section 3.2).

Conflicts may arise when two or more partners update the same piece of data without synchronizing between updates. When these pieces of data are synchronized, conflict resolution is needed (see section 3.3).

When the platform has been running for a while, it may become necessary to delete information items from the local storage. A cleanup process takes the IoI into account and then decides which items should be deleted (see section 3.4).

3.1 Discovery

Before synchronization between devices can occur, each device needs to be able to identify potential partners. DNS is typically used for translating host names into IP addresses (using A records) and vice versa (using PTR records), but it can also be used for service discovery with SRV records (Gulbrandsen et al., 2000). It provides a way for clients to query which servers provide a certain service, e.g. SIP, XMPP or - in this case - a synchronization service.

However, this discovery mechanism cannot be a centralized DNS system, because that would work badly when there is no network connectivity between a device and the DNS servers. Other systems exist that supply the same kind of functionality as DNS, but without the need for a centralized server, such as multicast DNS (Steinberg and Cheshire, 2005).

Using multicast DNS, the devices in the platform can query which other devices containing the platform are present in the network. Each device publishes that it provides a synchronization service that the other devices can use.

3.2 Synchronization

The information that is stored in the platform consists of the actual data the application wants to store and some pieces of meta data that are needed for synchronization. Figure 3 shows that there are four pieces of meta data:

- Universally unique identifier (UUID). This identifies the information in a unique way.
- Revision number. This identifies the revision of the information. If the information is changed, the revision number is incremented.
- Modification timestamps. This is an array of timestamps of all modifications of the application information. Its length is equal to the revision number.
- Saved timestamp. This contains the timestamp of the last change to this meta data, either due to a local change to the application data (and therefore an increase of the revision number) or because of synchronization with a partner.

The technique we present here is a slightly modified version of Multi-version Concurrency Control (Bernstein and Goodman, 1983). It adds the saved timestamp, which is used to speed up synchronization. A synchronization client knows when the last synchronization with a certain partner has taken place. It asks this partner for revision information since this timestamp. With the saved timestamp at its disposal, the server is able to respond with revision information that has actually changed since the last synchronization.

3.2.1 Platform pseudocode

The following pseudocode contains the high level synchronization code of the platform. The platform

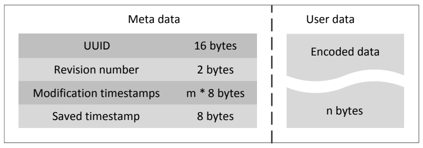


Figure 3: Information needed for synchronization (left side) and information stored by an application (right side).

switches between two states here, that of a client and that of a server.

```
1. Start up a synchronization server and publish its
   service through the discovery mechanism
2. Startup threads
3. Thread 0 (manage partner connections):
   (a) Detect new partners through the discovery
       mechanism
   (b) If a connection with a partner has been lost, try
       to reconnect
   (c) If reconnection fails too often, remove from
       partner list
   (d) When connected to a partner, start new synchronization thread
4. Thread 1 to n (synchronizing with partner):
   (a) Loop while connected:
       i. If in server state, start server code (see section 3.2.2)
       ii. If in client state, start client code (see section 3.2.3)
```

3.2.2 Synchronization server pseudocode

The following pseudocode contains the synchronization code for the server role.

```
1. Wait for a synchronization client to send commands and respond appropriately:
   (a) If GetRevisionInfo command is received, return
       a list of all revision info for the information
       items in the IoI of the client and that has been
       updated since the last synchronization
   (b) If GetItem command is received, return the
       whole information block for the information
       item as indicated by the UUID
   (c) If SwitchRole command is received, change the
       role to client and exit this server code
```

3.2.3 Synchronization client pseudocode

The following pseudocode contains the synchronization code for the client role.

```
1. Determine last timestamp of synchronization and
   local information of interest
```

2. Send the *GetRevisionInfo* command to receive the remote revision information of all items from partner since last synchronization and within local information of interest
3. For all remote revision information:
   (a) Determine local revision information based on remote UUID
   (b) Compare local with remote revision information and determine which case should be applied:
      i. If local and remote revision information are the same: do nothing
      ii. If remote revision information is a subset of local revision info: do nothing
      iii. If local revision information is empty: get remote application information with the *GetItem* command
      iv. If local revision information is a subset of remote revision info: get remote application information with the *GetItem* command
      v. If local and remote revision information are on different branches: conflict resolution is needed; use the *GetItem* command to get the remote application information and lookup the locally stored application information
   (c) Update saved timestamp in cases iii, iv and v
4. Determine information items with low rating that can be cleaned up (see section 3.4)
5. Change the role to server by issuing the *SwitchRole* command and exit this client code

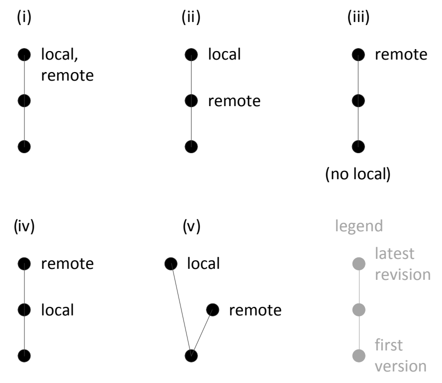


Figure 4: Revisions: do nothing (i and ii), get remote application information (iii and iv) and conflict resolution needed (v). Time flows from bottom to top.

Figure 4 shows the five options that exist for local and remote revision information. In options (i) and (ii) nothing needs to be done, because the local device already contains the newest information. In options (iii) and (iv) the remote information needs to be sent from the partner to the local device, because the remote information is newer or the local device does not have the information at all. Option (v) requires conflict resolution, because both the remote and the local device have changed the information since the last synchronization.

3.3 Conflict resolution

Systems such as CVS (Vesperman, 2006) and SVN (Pilato et al., 2008) show that it is possible to automate some kind of conflict resolution (merging), but also that manual intervention is sometimes necessary. In this paper we propose a system in which the application may choose to implement its own conflict resolution scheme, or default to the latest revision of the information.

It is crucial that the synchronization of information does not lead to a livelock, i.e. it should not lead to an endless synchronization loop when the information stays the same. The merged information when device A starts synchronizing with B must therefore be the same as the merged information when B starts synchronizing with A. This must also be true for more complex situations with three or more partners synchronizing with each other.

The left side of figure 5 shows how conflict resolution can lead to two different outcomes. The first outcome is the simple solution where the latest information is treated as the ”winner” of the merge (r3a is the latest timestamp). The second outcome is the more complex solution where the application decides to create a new version that is based on information of the older revisions (r4 is the latest timestamp).

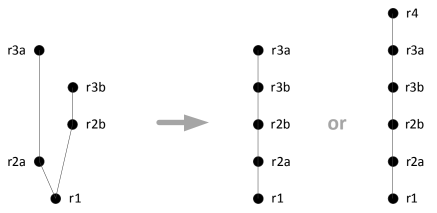


Figure 5: Conflict resolution leading to two possible outcomes: the latest revision wins (middle graph) or a new revision is created (right graph). Time flows from bottom to top.

When conflict resolution is needed, it is up to the application to decide which action to take. The platform provides an API that allows the application to merge conflicted revisions. It receives the revision information and historical application information of both the partner and itself, and is asked to provide either a new revision or agree to use the latest information, i.e. the information with the highest timestamp.

In the first case, it must ensure that a merge carried out by the same application on another device would lead to the exact same answer, otherwise livelocks might occur. In the second case, the platform handles the merge itself, of course without knowledge of the actual information involved, but with the guarantee that livelocks are avoided.

3.4 Cleanup

If an application decides to change the IoI or the storage capacity of a device is almost depleted, it may be necessary to perform some sort of cleanup.

If the IoI is decreased, the simple solution would be to just delete the information. However, this may result in information being deleted from the platform altogether. If the device still has storage capacity left, it retains this information until it actually runs out of storage capacity.

If the IoI is increased and the resulting information is too big for the local storage, the platform first checks if information was retained that is outside the IoI and deletes this. If this is not enough, the platform proceeds to delete information which is marked less important, such as older information or information located further away.

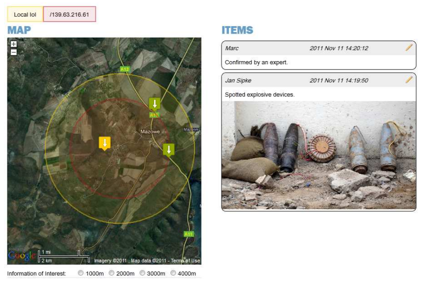
4 VERIFICATION

To verify that the storage platform is behaving as expected, a sample application was built that makes use of the platform for all its storage needs. Running this application shows that it is feasible to create applications using the storage platform and provides a means to test the platform manually.

Also, automated test cases were written which simulate that applications store new information, update it and synchronize it. The actual behavior was then checked with the expected behavior.

4.1 Sample Application

The application consists of a map where information items, e.g. text and pictures, have been pinned to a location. When a user clicks on a pin, the application shows all the items at that location. A selected item can then be edited by the user. If the user clicks on a new location on the map, a new item is created. All storage related tasks are handled by the storage platform. See figure 6 for a screenshot of the application. There are two versions of the application, the first running on regular personal computers and the second running on mobile phones and tablets.



Marc 2011 Nov 11 14:20:12
Confirmed by an expert.

Jan Sipke 2011 Nov 11 14:19:50
Spotted explosive devices.

Information of Interest: ○ 1000m ○ 2000m ○ 3000m ○ 4000m

Figure 6: Screenshot of an application using the storage platform.

4.2 Tests

Several manual and automated tests were performed to check that the storage platform is behaving as expected. The figures presented in this section show how information items progress in time when devices create, update and synchronize these items. In manual tests, the theoretical results were checked with the experimental results in the log files of the platform. In automated tests, these were checked automatically.

Figure 7 shows two devices updating a piece of information one after the other and synchronizing in between the updates. Figure 8 shows three devices where two devices update a piece of information without synchronizing in between. This conflict is resolved when all devices have synchronized with each other.

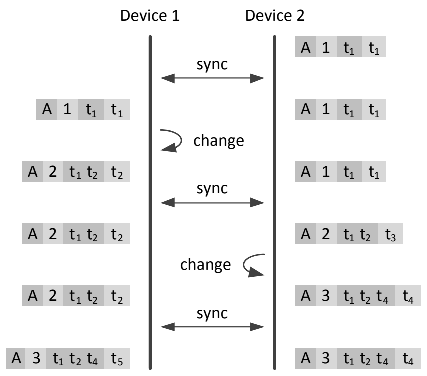


Figure 7: Synchronization test with two devices and no conflicts. Time flows from top to bottom.

5 FUTURE RESEARCH

The current storage platform has been implemented and tested on a small set of workstations, tablets and mobile phones. In future research we would like to test with many more devices and a much more diverse network setup, e.g. with real radio networks.

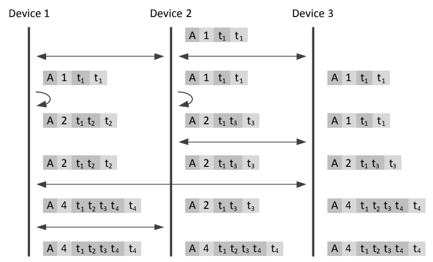


Figure 8: Synchronization test with three devices and a conflict, which is handled during the two synchronization steps at the bottom. Time flows from top to bottom.

Also more quantitative measurements will be made to assess the platform more thoroughly and compare it to existing distributed storage solutions.

At the moment, the application information blocks are quite small, upto a few kilobytes. If these information blocks are increased to several megabytes, it becomes necessary to synchronize only the parts that have actually changed, e.g. by using something similar to the rsync algorithm (Tridgell, 1999).

Some kinds of applications store hierarchical information, e.g. news postings with comments. At the moment, the application is responsible for storing the relationship between these information blocks. If the storage platform is aware of these relationships, it can make better decisions on their storage and possible deletion.

Finally, the current storage platform uses only time and location (i.e. a circle with a given center and radius on a map) to denote the IoI. Other forms of IoI are also interesting, such as author, clearance level and labels.

6 CONCLUSIONS

The mobile and dismounted domains of defense organizations cannot use standard cloud storage platforms because of unreliable network connections. This paper shows that it is possible to design a storage platform that does not depend on any centralized entity. Instead, it relies on a distributed discovery mechanism to find partners to synchronize information with.

The term information of interest was introduced, which is a way for applications to tell the storage platform which information is important. The platform uses this to decide what information to store and what to delete when storage capacity is depleted.

A sample application was built to verify that the proposed storage platform is a viable solution for the problems we described earlier. It has been shown to work on several devices, such as personal computers, mobile phones and tablets. With some manual and automated tests we also showed that the platform behaves as expected with regards to synchronization.

REFERENCES

Amazon (2011). Amazon simple storage service (s3). http://aws.amazon.com/s3.

Bernstein, P. A. and Goodman, N. (1983). Multiversion concurrency control - theory and algorithms. *ACM Transactions on Database Systems*.

Bohossian, V., Fan, C. C., LeMahieu, P. S., Riedel, M. D., Xu, L., and Bruck, J. (2001). Computing in the rain: A reliable array of independent nodes. *IEEE Transactions On Parallel And Distributed Systems*.

Brewer, E. A. (2000). Towards robust distributed systems. *Symposium on Principles of Distributed Computing*.

Burbank, J. L., Chimento, P. F., Haberman, B. K., and Kasch, W. T. (2006). Key challenges of military tactical networking and the elusive promise of manet technology. *IEEE Communications Magazine*.

Gulbrandsen, A., Vixie, P., and Esibov, L. (2000). A dns rr for specifying the location of services (dns srv). http://www.rfc-editor.org/rfc/rfc2782.txt.

Halinger, G. and Hohlfeld, O. (2010). Efficiency of caches for content distribution on the internet. *Teletraffic Congress*.

Hancock, P. A. and Szalma, J. L. (2008). *Performance Under Stress (Human Factors in Defence)*. Ashgate.

Nguyen, T. T. M. and Dong, T. T. B. (2011). An adaptive cache consistency strategy in a disconnected mobile wireless network. *IEEE International Conference On Computer Science and Automation Engineering*.

OpenStack (2011). Openstack object storage. http://openstack.org/projects/storage.

Pilato, C. M., Collins-Sussman, B., and Fitzpatrick, B. W. (2008). *Version Control With Subversion*. O'Reilly Media.

Rhea, S., Wells, C., Eaton, P., Geels, D., Zhao, B., Weatherspoon, H., and Kubiatowicz, J. (2001). Maintenance-free global data storage. *IEEE Internet Computing*.

Shinkuma, R., Jain, S., and Yates, R. (2011). In-network caching mechanisms for intermittently connected mobile users. *Sarnoff Symposium*.

Steinberg, D. and Cheshire, S. (2005). *Zero Configuration Networking: The Definitive Guide*. O'Reilly Media.

Suel, T., Noel, P., and Trenafilov, D. (2004). Improved file synchronization techniques for maintaining large replicated collections over slow networks. *International Conference on Data Engineering*.

Thomas, T. L. (2000). Kosovo and the current myth of information superiority. *Parameters*.

Tridgell, A. (1999). Efficient algorithms for sorting and synchronization. PhD thesis, Australian National University.

Vesperman, J. (2006). *Essential CVS*. O'Reilly Media.