INTRODUCTION

1 An Introduction to Data Synchronization

Mobile computing gives everybody access to their business or personal data everywhere, using devices like Personal Digital Assistants (PDAs), smart phones, mobile phones, and laptops. An online connection to a corporate datastore might not always be possible, due to lack of network coverage, for example. Sometimes even if a connection is available, using it might not necessarily be the fastest and most cost-effective way for the application to operate. In situations such as these, data synchronization is a key technology to alleviate those shortcomings.

Data synchronization allows a consistent local "copy" of various kinds of data, from a central corporate datastore or a service provider datastore on the user's device. It is therefore possible to look up or change data locally on the device without requiring an online connection to the master copy of that datastore. The simplest case is a user retrieving data for his local copy. Here, the application needs to get only the changes from the master datastore to the local copy, without copying the complete datastore again. Synchronization gets more complicated as soon as a lot of different users make modifications to their local copies of the datastore. Now, somehow, these modifications need to be reconciled between all copies of that datastore.

Data synchronization is the technology used to keep all these distributed copies of a datastore consistent by communicating the actual changes between these copies and by resolving conflicts that may arise due to contradictory changes in different copies of the same datastore.

Today, synchronization services support Personal Information Management (PIM) data, such as addresses, calendar entries, memos, and to-do's, as well as Relational Databases and file systems.

The following paragraphs describe the different possible synchronization topologies: one-to-one, many-to-one, many-to-many, and two hybrid versions. These definitions are followed by explanations of the different synchronization modes: local, pass-through, and remote.

The different challenges and problems that arise while keeping data synchronized are elaborated in the following part. This chapter closes with an overview of related standards organizations, most of which have chosen to change their synchronization technology in favor of SyncML®.

The Different Topologies

Changes made to different copies of a datastore can be propagated to other copies of that datastore in different ways. The synchronization topology defines the logical flow of the changes propagating through the network of computers hosting instances of that datastore. The four major topologies are:

- One-to-one
- Many-to-one
- Many-to-many
- Hybrid of many-to-one and many-to-many

One-to-One

The one-to-one topology is the simplest case. The other topologies can be seen as an extension of this one. Here the data is only shared between one server (the square in Figure 1–1) and one client (the circle in Figure 1–1). A possible usage scenario for this topology is a datastore that is mirrored for backup purposes. All changes made to the client are

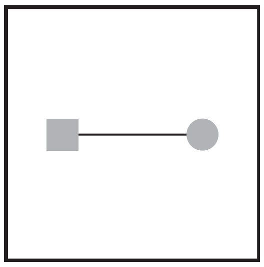

Figure 1–1
One-to-one topology

also sent to the server to ensure that its copy of the data reflects the current version of the client copy. Assuming that data is only changed in the client directly (i.e. no modification is made to the server copy besides synchronizing with the client), then there is no risk of any conflict in this topology. The one-to-one topology is also known as the "Dedicated Pair" topology.

This kind of topology is also used between someone's PDA and personal computer, with the difference that changes are usually made on both the PDA and the personal computer. In this case, the conflicts are typically identified by the PC and directly resolved on the PC. In some cases the conflict is marked and the user is asked to resolve it.

Many-to-One

Numerous commercial systems are examples of the many-to-one topology (also known as central master or star topology). In this topology, data is propagated from a central master to the different entities containing copies of the data, as shown in Figure 1–2.

The main advantage of many-to-one topology is its relative simplicity to implement compared to many-to-many topology, which is described in the next section.

All clients exchange data with the central server only–two clients cannot exchange data directly without the intermediary central server. Because of this characteristic, conflicts can only arise at the central server, which needs to detect and resolve them. The clients themselves do not need to worry about conflicts. They just inform the central master about the local modifications and process the change requests they receive from the central master. There is no need for the client to determine where to send it, as in the many-to-many topology.

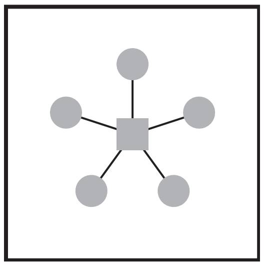

Figure 1–2
Many-to-one topology

This topology is common when a person has a PDA, a cellular phone, and a personal computer sharing an application such as the calendar application, and both the cellular phone and the PDA are synchronized with the personal computer (but not between themselves). This kind of interaction is also common when family members carry cellular phones and update their shared family Web calendar independently or when mobile employees in an enterprise update inventory datastores independently.

The drawback of this architecture is that the central master could become a bottleneck, a single point of failure that could immobilize the entire system. Let's consider an Internet service provider scenario with a central master that serves several hundred thousand accounts, all trying to synchronize with the same central datastore. Here the central master should not be a single server, but a cluster of high-performance servers to limit the latency in response time even if one of the servers fails.

Many-to-Many

In many-to-many (or peer-to-peer) topology, there is no central server. Every client is also a server, as shown in Figure 1–3. For simplicity in this chapter, the client/server combination on each device in the many-to-many topology is just called client.

Every client gets updates from and sends updates to every other client. After a record on one client is updated, this client is responsible for updating all the other copies of the data on all the other clients to ensure that the consistency of the distributed datastore is maintained. This might be by directly contacting the other clients or by sending the updates to the clients nearby, which are then responsible for propagating it further.

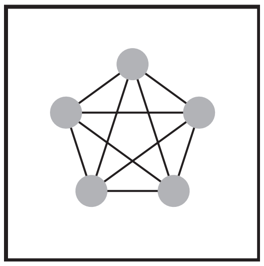

Figure 1–3
Many-to-many topology

Consequently, every client must be able to detect and resolve conflicts. This requires more complex software on each client, which naturally increases the implementation cost, especially on small mobile clients, like mobile phones, in which memory is a scarce resource.

Compared to the many-to-one topology, the many-to-many topology is more robust but also clearly adds to the complexity. In this topology, it is very difficult to find out if a modification was indeed propagated to all clients at a given point in time.

One advantage of the peer-to-peer topology is that without a central server, there is no single point of failure. Every client has a copy of the data and can act as a server. The clients can continue to work and exchange data despite failures in other parts of the network. A client can retrieve updates from the closest server in the network, which gives quicker access to data otherwise stored remotely.

This topology may occur whenever there is no notion of a primary datastore involved in the system. Consider a team of emergency response workers taking readings such as measured temperatures, toxin levels, and structural stress conditions in a building or an affected area. They can synchronize these readings as they pass by each other using direct wireless or infrared links between their handheld devices.

Hybrids of Many-to-One and Many-to-Many

In an effort to combine the advantages of many-to-one and many-to-many topologies, hybrids containing characteristics of both types can be used, as shown in Figure 1–4.

The cluster consists of a two-level structure of data copies. The top level consists of a cluster of servers. All servers contain copies of the data and replicate between each other, but for each data object only one

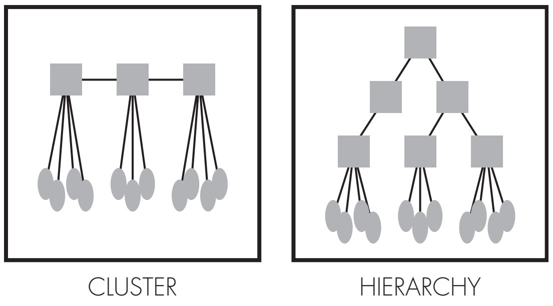

Figure 1–4
Hybrids of many-to-one and many-to-many topologies

server keeps the authoritative copy. The other servers are unaffected by the failure of one of them. Using geographically distributed servers can contribute to reducing the distance between server and clients.

In a hierarchy, the server structure could be modeled according to the organizational structure of a company. The top part of the figure shows servers, which are at the same time clients of a server one level above. In this structure, even when one section experiences a failure, the overall topology can still work properly.

In commercial implementations using a central master topology, the master server itself consists of a cluster of servers accessing a central datastore. This setup guarantees high availability and reduces the disadvantages of a central master topology with regard to the single point of failure. Nevertheless in this setup the servers are physically at the same location and a network failure could make them unreachable. That would not be the case in the cluster or hierarchy topology, as described above.

Summary

Clearly, the one-to-one and the many-to-one topology are abundantly more common in the context of day-to-day commercial applications than the many-to-many topology. Moreover, the many-to-many topology can be indirectly (but inefficiently) achieved by designating one device as a server and stipulating that the other devices synchronize with that server and hence indirectly synchronize with each other via that server. Implementing the many-to-one topology (which includes the one-to-one model) is conceptually simpler and the resulting implementations are orders of magnitude simpler than the ones that support many-to-many topology. In the many-to-many model, complex data structures such as "version vectors" need to be associated with data items to correctly synchronize data. The many-to-many model is also especially stubborn for the purposes of accounting and failure recovery.

For the above reasons, SyncML is optimized for the many-to-one topology. It allows the exchange of datastore sync anchors (see Chapter 5) in the beginning of a synchronization, which indicate the last "timestamp" at which the two computers synchronized. The timestamp could be an actual time value or a logical counter. Based on the exchanged sync anchor values, the associated sync engines could use simple data structures such as change logs (see Chapter 4) to determine the changes made to data items since the last instance of synchronization. SyncML, however, *allows* many-to-many synchronization. It allows each data item to have an associated version which could actually be a version vector required for many-to-many synchronization. It also does not specify the format of the sync anchor explicitly and therefore that could also be a version vector. Furthermore, a SyncML device can play dual roles of a server or a client.

The Different Usage Modes

Synchronization is typically used in the following three physical ways or modes: local, pass-through, and remote. Each synchronization mode has its specific challenges, which are described below.

Local

This is currently the most common way to use synchronization. A typical scenario, shown in Figure 1–5, has a user synchronizing data from the PC or laptop on his desk with a PDA that is connected to the PC or laptop through a serial, Universal Serial Bus (USB)™, infrared, or Bluetooth™ connection.

An application running on the PC acts as the synchronization server. The data is usually retrieved from and stored in another application on the computer, such as Lotus Notes® or Microsoft Outlook®. This scenario requires the user to first synchronize his desktop application with the server (e.g. Lotus Domino™ or Microsoft Exchange®) to retrieve the latest updates from the server before synchronizing the PDA with the desktop.

Pass-Through

This scenario can also be called "router." In this scenario, the synchronization client is connected locally to a computer, similar to local

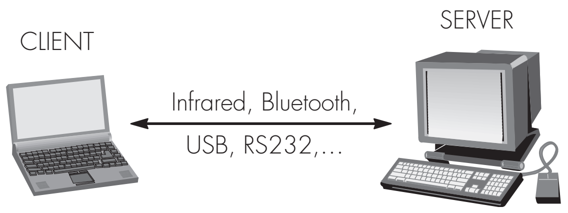

Figure 1–5
Local synchronization

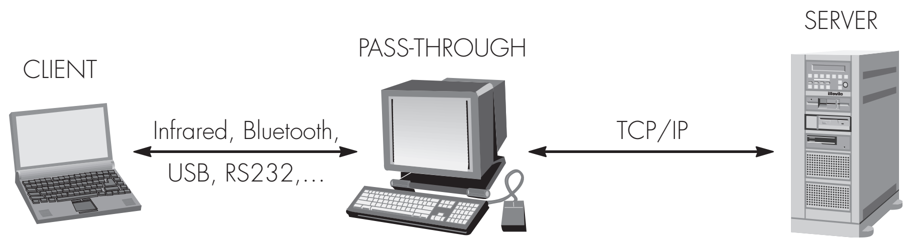

Figure 1–6
Pass-through synchronization

synchronization. But in this usage mode, the local computer is used as a router to pass the synchronization requests to another server acting as the "true" synchronization server. In this usage mode, shown in Figure 1–6, a device can synchronize with a remote server thanks to the pass-through service on the local computer bridging the technologies, even if this device only supports infrared or Bluetooth connectivity.

While in the office, the device can use the faster and cheaper network connection between a desktop computer and the server. A drawback is that the synchronization is now done with a remote server, even while in the office, which generates some additional network traffic compared to the local synchronization case. While on the road, the user uses the remote synchronization to directly synchronize with the synchronization server, as described in the next paragraph.

Remote

Remote synchronization, shown in Figure 1–7, is used if the synchronization client is connected to the synchronization server using a network connection. This might be a local area network (LAN) or a wireless network like GSM (Global System for Mobile Communications) or UMTS (Universal Mobile Telecommunications System).

With wireless networks like GSM, though, the data exchange rate available is obviously not as high as when connected to the synchronization server by a LAN. The advantage of a PDA and/or a mobile phone is that the user can access the server and synchronize the data from almost everywhere in the world.

PH052-Hansmann01.fm Page 11 Tuesday, August 27, 2002 11:00 AM

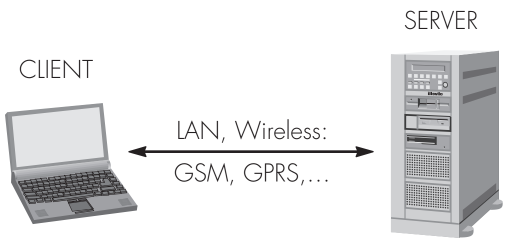

Figure 1–7
Remote synchronization

Challenges with Data Synchronization

Data synchronization technology is a very important part of enabling offline usage of data, but it also presents some challenges that need to be solved in order to provide a sound solution. The following section focuses on the central master topology, since it is currently the most widely used.

The main challenges are:

- Conflicting updates
- Conflict detection
- Conflict resolution

Another challenge with data synchronization is the need to agree on a common content format for data exchanges. For PIM data, SyncML requires the support of specific content formats to ensure successful synchronizations.

Conflicting Updates

In Data Synchronization, where copies of the same data reside on more than one device, it is very likely that the same entry is manipulated on more than one device.

Consider a datastore with customer records, containing the customer's name, address (street and city), and data about recent orders. This data is centrally stored in the corporation's mainframe computer. The salesmen who visit customers during the day synchronize their PDAs with this corporate datastore every morning and every evening.

During his visit at a customer, the salesman changed the fax number for the customer "Smith" to "12345". The same day, the customer relations department received a letter from Mr. Smith notifying them that his fax number has changed. The customer relations agent updated

Mr. Smith's record, but unfortunately made a typo while entering the number and updated the entry with "92345".

Two conflicting updates to the same record have been made since last time the salesman synchronized the customer datastore. It is now up to the synchronization server to first detect the conflict and then resolve it.

If the synchronization server could not detect the conflict, it would send the update it had received from the customer relationship department to the salesman's PDA and would also update the master datastore with the update the salesman made during the day. As a result, the content of the two copies would not contain the same data–the master datastore would have the fax number "12345", and the salesman's PDA would contain "92345". This problem would probably remain, undetected and uncorrected, until somebody updated the same object again. The problem described below is illustrated in Figure 1–8.

The update conflict described above is known as a *write-write conflict*: The same field is updated in the same time period in two different

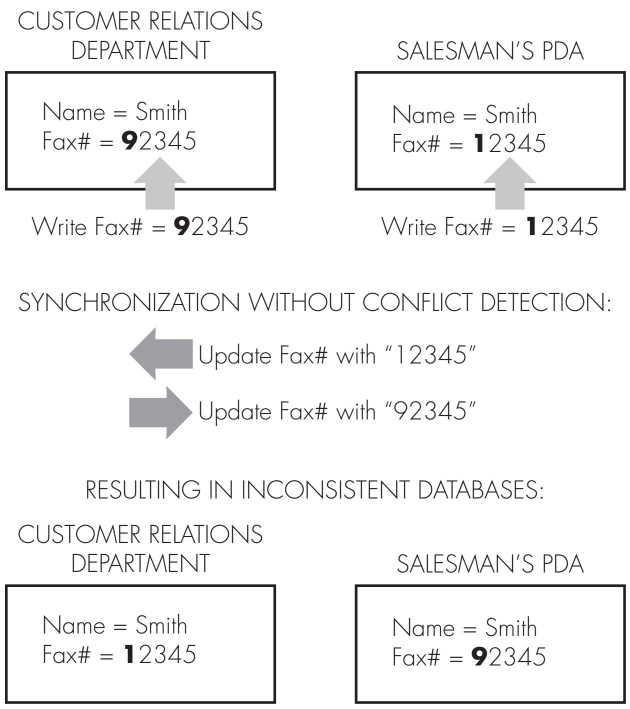

Figure 1–8
Synchronization without conflict detection

copies of the datastore. Other update conflicts include *read-write conflict* and *constraint violations*. They are explained later in this chapter.

For example, examine the following scenario: During the day a salesman is entering a new order for the customer with the name "Smith" and changes his fax number from "98765" to "12345". At the same time, the corporate marketing department is creating a list of all customers' fax numbers for a mass mailing. The salesman hasn't synchronized his PDA yet, and therefore the central datastore's entry for the fax number of the customer "Smith" still contains "98765", where "12345" is the correct value. This situation is called a *stale read*, where one retrieves an entry from one copy of the datastore that is not up to date because a change was made to another copy of the same datastore and wasn't propagated to all other copies of the datastore. Figure 1–9 illustrates this situation.

In the case of a read-write conflict, the update to a data field is made based on the value of another field in the datastore. While making this update in one copy of the datastore, the field that determined the decision to update the other field was changed in another copy of the datastore. Therefore the update is made based on a stale read. Detecting these conflicts is especially important for Relational Databases. PIM systems rarely have these conflicts.

One possible example of a conflict based on a constraint violation is where a certain field in the datastore is constrained to accept only values

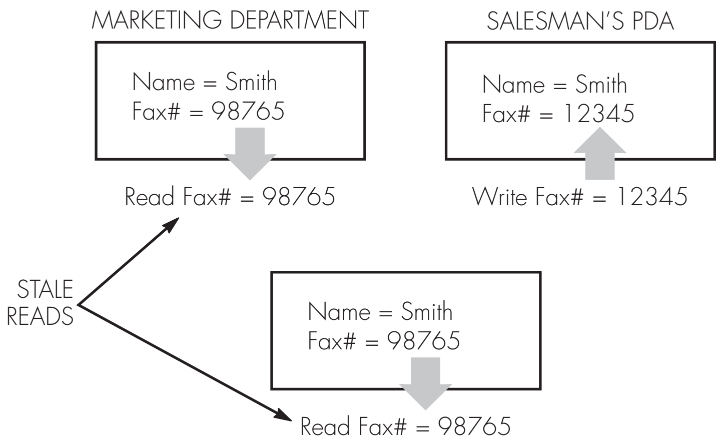

Figure 1–9
Stale reads

between “1” and “10”. Now this constraint is changed such that only values between “1” and “5” are acceptable. In another copy of the datastore, somebody enters a value of “8” into the field with this constraint. During the next synchronization the synchronization server has to detect this conflict if the datastore is to be kept consistent.

One possible way of preventing conflicting updates (especially read-write conflicts) is to obtain exclusive access to an object before updating it. In the central master scenario this would require the device to contact the central master and request exclusive access. This might be possible in intranet scenarios, but is not an option with mobile devices. Mobile devices might be in areas that currently have no network coverage. Also, setting up a wireless network connection, for example via GSM, takes a few seconds and can be quite expensive (especially when roaming internationally). Therefore getting exclusive access to the object before updating it is not a practical alternative. Thus, a synchronization server must be capable of detecting and resolving conflicts.

The conflicts mentioned above all belong to a category of conflicts called *mechanical* conflicts, since there have been concurrent modifications to the same record. It is also possible, however, for a user to enter a meeting with a customer from 8:00 to 10:00 in the morning while his secretary blocks the 9:00 to 10:30 slot for a meeting between the user and his manager. From a synchronization technology point of view, there is no conflict. Both records were newly added distinct calendar entries. But the user now has to attend two meetings from 9:00 to 10:00, which is a *semantic* conflict that the user needs to solve. The calendar application can in this case support the user by detecting the conflict and informing the user about this.

Conflict Detection

Conflict detection is important in order to keep different datastores in “sync,” meaning keeping them consistent.

One important prerequisite of being able to synchronize data is the ability to uniquely identify each record in the datastore. Relational Databases have primary keys that serve this very purpose. In other types of datastores, such as PIMs for example, UIDs (unique identifiers) are used. The UIDs in the datastore on the central master are usually called GUIDs (globally unique identifiers; possibly only unique in the scope of that one datastore). Each copy of the datastore on a client has

PH052-Hansmann01.fm Page 15 Tuesday, August 27, 2002 11:00 AM

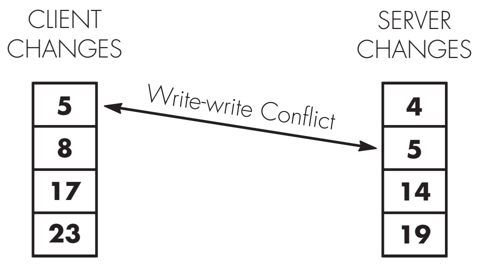

Figure 1–10
Identifying an update conflict

its own UIDs to identify the records, which are called LUIDs (locally unique identifiers).

The central master maintains a datastore to store the LUID of each record for every client that synchronized a given datastore. While synchronizing, the central master uses this LUID to tell the client which record to modify.

A synchronization session usually starts with the client sending a list of changed records since the last synchronization for a particular datastore to the server. Next, the central master then generates a list of all modifications that occurred during in the same time period. To detect possible conflicts, the server then compares the two lists and identifies every LUID/GUID combination that exists in both of the lists as a conflict, as shown in Figure 1–10.

Conflict Resolution

Conflict resolution is the action that the server must take after a conflict has been detected.

There exists a wide variety of methods a server can use to resolve conflicts. The easiest way for the server is to duplicate the entries and to mark them as conflicts. This strategy does not lose any data, but puts the burden of resolving the conflicts on the user (who has to manually choose the correct record or merge the two records).

The next possibility is for the server to identify one of the two conflicting records as the winner and delete the other one. The decision can be based on user preferences, such as:

- Updates made on the client always win.
- Updates made on the server always win.
- The latest change wins.

Basing the decision on where the change was made is a pretty simple method to implement. Making the decision based on which one of the two conflicting updates was the latest one is more complicated. First, this requires the datastores to record the time when the change was made. This is often impossible, especially with mobile devices, like mobile phones, or PDAs, like the Palm Pilot™. For this to work, the system time needs to be synchronized across all the devices involved, or a scheme must be set up to address time-drift.

The third method to resolve conflicts is to merge the two conflicting records into one new one. This could be done in cases where complete records are synchronized, for example an address book record containing name, street, city, and phone number. Additionally, this requires the synchronization server to understand the structure of the data that is synchronized and to be able to identify the field within that structure that was changed. This is not possible if the records contain only single fields that are to be synchronized, and the synchronization server cannot interpret the data in the field.

Resolving conflicts with PIM data is relatively easy. It is more complex with Relational Databases, which usually have relationships between different records that need to be honored during conflict resolution.

Related Work

Synchronization is an important component of many standards for mobile and wireless technology. The SyncML Initiative is closely working with other organizations, especially Infrared Data Association (IrDA®), Wireless Application Protocol (WAP) Forum™, and Third Generation Partnership Program (3GPP)™, to get them to agree that SyncML (Data Synchronization and Device Management) should be the synchronization technology of choice in their specifications.

Infrared Mobile Communication

The IrDA mobile communications committee defined the Specification for Infrared Mobile Communications (IrMC) [IrMC00], to provide information exchange over infrared. IrMC defines five levels of information exchange:

- Level 1 (Minimum Level)
- Level 2 (Access Level)
- Level 3 (Index Level)

- Level 4 (Sync Level)
- Level 5 (SyncML Level)

Level 1 to Level 4 of the IrMC specification define the exchange of a limited number of different objects, such as business cards, calendar data, messages, and email data over personal area networks with connection-oriented or connectionless links. They do not support the synchronization of other forms of data such as Relational Databases or tabular data. Since the IrMC specification was initially designed for local data synchronization, the methods proposed are not optimized for data synchronization over wide area networks, such as synchronizing the phone book on a mobile phone with one of the corporate public address book datastores over the Internet.

Level 5 was added in 2001 and defines SyncML as the synchronization technology used in IrMC. The SyncML Initiative has closely worked with IrMC in order to produce a Synchronization Profile that strongly recommends SyncML as the preferred synchronization technology.

Wireless Application Protocol

In June 1997, Ericsson®, Motorola®, Nokia®, and Unwired Planet® (now Openwave®) founded the Wireless Application Protocol (WAP) Forum as an industry group for the purpose of extending existing Internet standards for the use of wireless communication. By spring 2002, the WAP Forum® had more than 500 member companies from all parts of the industry, including network operators, device manufacturers, service providers, and software vendors.

The WAP Specification Version 1.1 was released in the summer of 1999, and the first WAP devices and services were available as early as the fourth quarter of 1999. Version 2.0 of the WAP Specifications was approved and released to the public in February 2001.

Since version 2.0, WAP requires all WAP Servers and WAP Client Devices supporting data synchronization to use SyncML. WAP also requires clients and servers to pass the SyncML Conformance testing as a prerequisite for passing the WAP Conformance testing.

The Open Alliance® was formed in June 2002. OMA was created by the consolidation of the supporters of the Open Mobile Architecture® initiative and the WAP Forum. Additionally, SyncML Initiative® Location Interoperability Forum (LIF), MMS Interoperability Group (MMS–IOP), and Wireless Village® announced they had signed

Memorandums of Understanding of their intent to consolidate with OMA. This consolidation should take place by the end of 2002.

Third Generation Partnership Program

The Third Generation Partnership Program (3GPP) was established in December 1998 as an organization of all partners interested in the evolution of mobile systems to the third generation evolving from the GSM technology. GSM is looked at as the second generation, GPRS (GSM Packet Radio Service) and EDGE (Enhanced Data Rates for GSM Evolution) as the 2.5 generation, and UMTS, UTRA (UMTS Terrestrial Radio Access), W-CDMA (Wideband Code-Division Multiple Access), and FOMA (Freedom Of Mobile multimedia Access) as the third generation of mobile communication technology.

SyncML and 3GPP are working together closely, and SyncML technology has been mandated as the method of choice for data synchronization since Release 4 of the 3GPP specifications.