Software Platforms for Mobile Distributed Systems

Rakesh Polepeddi¹, Koushik Vanamà²

Green Fields, Vaddeswaram.

Abstract: As a result of the computing technology that becomes ever smaller and cheaper it is now possible to integrate it into everyday material objects. This advanced integration of technology allows the underlying computer to disappear into the fabric of life so that by manipulating material objects we are transparently interacting with the underlying integrated technology. The invention of wireless communication technology enables these disappearing integrated computers to cooperate with one another so that they can derive context about its environment. The advantage is that users can be supported more naturally and transparently to achieve their goals. This vision is often referred to as “Ambient Intelligence” (AmI). The research presented in this dissertation deals with the problem of software development for these invisible computers from the perspective of distributed systems.

Keywords: distributed systems, concurrency distribution, library approach, integrated approach.

I. INTRODUCTION

Much of the emerging behavior in ubiquitous computing scenarios results from the cooperation between devices. These devices can cooperate because they are surrounded by what is sometimes referred to as a mobile network. A mobile network emerges from a set of devices that communicate over wireless communication media. The systems that result from such a hardware constellation are called mobile distributed systems. A mobile distributed system explicitly supports mobile computing. Mobile computing concerns the computation that is carried out in mobile devices. Mobile computing should not be confused with mobile computation, which concerns the mobility of code between devices. In this dissertation we focus on mobile computing. The evaluation of distributed programming platforms in the context of mobile distributed systems. One of the evaluation criteria is how they integrate with the object-oriented paradigm.

II. TYPES OF MOBILE DISTRIBUTED SYSTEMS

In this section we examine the commonalities and the differences of fixed and mobile distributed systems by means of a conceptual framework.

Definition (Distributed System)
A distributed system consists of hardware and software components located at networked computers that communicate and coordinate their actions only by message passing. From this definition we can zoom in on three facets of distributed systems:

• Type of Device: In the definition above the term “networked computer” can refer to a fixed device or a mobile device. Fixed devices range from desktop computers and server racks to electronics embedded in stationary objects such as a washing machine. On the other hand, mobile devices can vary between laptops, PDAs, mobile phones and other electronics embedded into mobile items, such as a wrist watch.

• Type of Network Connection: The word “communication” refers to the network infrastructure and this is the basis for another difference between fixed and mobile of distributed systems. On the one hand, in fixed distributed systems computers are often connected via permanent links. These links are often high bandwidth and supported by redundant infrastructure such that connections are relatively stable. Hence, disconnections are either caused by scheduled maintenance or unforeseen failures. On the other hand, mobile distributed systems are usually connected via a wireless communication link over wireless technologies such as Bluetooth, Wireless Fidelity and GPRS. These wireless technologies are prone to disconnections due to the limited communication range of these technologies. When users move about with their mobile devices they leave and enter the communication range of other devices in the environment, but even when two wirelessly communicating devices are stationary the link can be broken due to a radio occlusion caused by the environment, such as a car that passes in between the two communicating devices. The communication...
range is often further reduced by the limitations of the power source. The general rule is: the less power is available for the wireless link the smaller the communication range of the wireless link. Of course there are other issues that can greatly influence the quality of the wireless link such as the type of antenna that is used. An example of this is the quality of conversations over a mobile phone which are at times problematic even though there are a great number of antennas posted throughout many cities nowadays. Another source for disconnections are caused by the use of a finite power source in a mobile device. When a battery of a mobile device is discharged then the device stops functioning and active connections are lost or wireless links may be manually or automatically turned off to conserve battery power. From this we can conclude that mobile distributed systems are intermittently connected as opposed to fixed distributed systems that usually have permanent links.

- **Type of Execution Context**: Another facet that is maybe less explicit in the definition above is the execution context of a distributed system. With the term “execution context” we refer to the context information that can influence the behavior of an application. Typically in fixed distributed systems the execution context is more static than with the mobile variants. For example, the quality of a connection can depend on the environment in which mobile devices communicate while the quality of a connection in a fixed distributed system is often continuously stable. Another important type of execution context that is influenced by the location of mobile devices is the availability of services. In mobile distributed systems the availability of services often coincides with the location of the mobile device, whereas in a fixed distributed system services are often continuously available for an application.

Ubiquitous computing scenarios entail that computing technology is embedded in all types of devices, ranging from washing machines and refrigerators to cars, clothes and wrist watches. It is clear however, that most of the cooperation between these devices will occur over wireless communication media. Namely, wireless communication media makes the users oblivious to the computing technology in the face of mobility. Based on this conceptual framework of distributed systems we can further distinguish between two types of different mobile distributed systems:

- Nomadic distributed systems have a mix of fixed and mobile characteristics. A nomadic distributed system is built out of fixed and mobile devices that interact and cooperate via infrastructure. This infrastructure can be composed of wireless access points that are themselves connected via a fixed network. An example of such a distributed system is a mobile phone network, where each phone connects to an antenna and the different antennas are connected via cables. As users move about with their mobile phone the connection is transparently carried over from one antenna to another.

- Ad-hoc mobile distributed systems consist of a set of mostly mobile devices that are connected via extremely variable quality links and execute in dynamic environments. For example, mobile devices can be completely isolated from other devices and groups of communicating mobile devices may spontaneously emerge in the environment. Ad-hoc mobile distributed systems further distinguish themselves from their nomadic variants in that there is no infrastructure that supports the communication between devices. Such a network that emerges due to the mobility of the mobile hosts is often called a mobile ad-hoc network.

Both types of mobile distributed systems, discussed above, can be used to realize ubiquitous computing scenarios. For example, nomadic distributed systems can be useful to realize ubiquitous computing scenarios in the context of a restricted environment such as an office space or at home. Nevertheless, the vision of ubiquitous computing is not a delimited concept that starts in a restricted environment and stops when you leave it. For this reason ad-hoc mobile distributed systems are needed to further support the scenarios that continue outside of restricted environments such that no assumptions on the available infrastructure can be made.

### III. CONCURRENCY AND DISTRIBUTION

Although concurrency and distribution are theoretically not the same, the implementation of a distributed system is almost always concurrent. As a consequence a good concurrency model is the foundation of a model for distribution. In this section we review these aspects in the light of mobile distributed systems and the subsequent hardware phenomena. This insight is important because it influences design decisions of the distributed languages and middleware that are discussed in the subsequent sections.

#### A. Denoting Parallel Units in Programming Languages

A first important concept that we find in software is the ability to spawn parallel activities. A parallel activity is expressed as a parallel unit. Such a unit can
range from a process to the level of expressions.

1) Processes and Threads: Processes are perhaps the most frequently used unit of parallel activity. Many operating systems run each program in a separate process. Processes have their own state and data, hence no memory is shared between processes. Within such a process it is possible to create multiple threads. In contrast to processes, threads can share memory and allow for more fine-grained parallel activity. In most mainstream languages threads are created dynamically and terminate when the top-level procedure they are executing returns. However, often some functionality is provided to abort a thread such that it terminates before this top-level procedure has returned.

2) Objects: There are several options to introduce parallel activity onto objects. In a sequential object-oriented programming language objects interact via message passing. An object sends a message to itself or another object and waits until the receiving object has processed the message and returns control back to the sender. This is often paired with a value that is returned to the sender. Four different options to map parallelism onto an object:

   - Attach a thread to an object and the object can be active without having received a message.
   - Allow the object to continue its execution after it has sent the message. In other words, an object sends a message and continues its execution without waiting for the receiver to have completed processing the message.
   - Instead of sending the message to a single destination the message can be sent to multiple objects that each process the message in parallel. The sender waits until the different receivers have finished executing the message.
   - The receiving object continues executing after it has returned control to the sender.

3) Expression and Statements: The most fine-grained unit of parallelism is expressed at the level of expressions and statements. For example, in Occam it is possible to declare the parallel execution of a number of statements using the PAR keyword. Parallelism at the level of statements or expressions is easy to understand and use but difficult to maintain in large applications, because it is more fine-grained.

B. Design Issues in Communication

After having discussed the different ways of introducing parallelism in programming languages we can now turn our attention to the way one logically distributed parallel unit communicates with another by message sending. There are four important characteristics that must be considered for the communication between parallel units in the context of distributed systems. These characteristics are further discussed below.

Characteristic #1: Addressing Parallel Units A first consideration is how to address distributed parallel units. Addressing a parallel unit can be either direct or indirect. A parallel unit is addressed directly when its communication partner addresses it explicitly. An example of this explicit addressing are remote object references, where an object is directly referred to by another object. On the other hand, indirect addressing of a parallel unit occurs when its sender does not refer to it directly, but instead refers to an abstract intermediary communication partner that in its turn refers to the parallel unit that needs to be addressed. Note that indirect addressing offers greater flexibility and a higher level of abstraction that can be useful to deal with the ever-changing environment in which mobile devices are used.

Characteristic #2: Implicit vs Explicit Communication Once a parallel unit can be addressed it is possible to communicate with it. A parallel unit sends a message that is received by another parallel unit. Sending a message is usually always explicit in the code, but receiving a message can be either implicit or explicit. For example, when a method is invoked on a remote object in Java RMI, then this method invocation is implicitly accepted. The object does not have to accept the message explicitly. Occam is a language where messages are accepted explicitly. In Occam syntax is provided such that a process can explicitly listen to a channel for a message.

Characteristic #3: Communication Timing Another important decision is whether to use synchronous or asynchronous communication between parallel units.
Definition (Synchronous Communication) The sender and the receiver both synchronize at every message.

Definition (Asynchronous Communication) The sender does not wait for the receiver to be ready to accept its message.

- Asynchronous communication with rendezvous: The sender of a message blocks until it has received an acknowledgment from the receiver that is has been received. However, the sender does not wait until the message has been processed.

- Asynchronous communication with FIFO order: The sender of a message is guaranteed that the messages it sent to a receiver are received in the order it has sent them.

- Asynchronous communication without order guarantee: Messages sent by a parallel unit can be received in any order, irrespective of the order in which the messages were sent. An example of a low level messaging protocol with such semantics is UDP.

The latter two types of asynchronous communication match well with inherently asynchronous distributed systems such as the ones found in mobile distributed systems, because they allow one to abstract from unavailable devices such that the autonomous nature of devices is not hampered.

Characteristic #4: Reliability There are several degrees of reliability that can be guaranteed when communicating. For example, on the one end when communicating using UDP there are no delivery guarantees made to the sender of a message. If a message is lost in transit it will never arrive at its destination. On the other hand, the sender of a message can continuously retry sending a message until it receives an acknowledgment that the message has been received. Between these two extremes there are approaches that provide some fault tolerance to a limited extent. An example of this is the TCP/IP protocol, which is frequently used for connections over the internet. In any case, there is no single strategy that is suitable for all distributed applications. However, in the case of mobile distributed systems, where volatile connections are the rule rather than the exception care must be taken in making the correct choice. Note that the degree of reliability is independent from the communication timing. On the one hand, synchronous communication can be made reliable by blocking the sender and meanwhile retrying to send the message. On the other hand asynchronous communication can be made reliable by transparently resending a message until an acknowledgment has been received.

C. Corollaries of Mobile Distribution

Above, we have discussed fundamental concepts found in distributed systems and how these concepts can be translated into programming language concepts. We can now revisit the consequences of these choices in the context of mobile distribution.

1) Non-deterministic interactions: Non-deterministic interactions are a distinct characteristic of distributed systems and are a consequence of the use of multiple independent machines (which have their own internal clock and speed) that are acting on a shared resource. An important insight is that the type of communication determines to a large extend the degree of non-determinism that can occur. For example, suppose a distributed bounded buffer object. The buffer is accessed by a single producer and multiple consumers. This example can be used to compare synchronous communication with and without an explicit receive statement. One problem is what happens if the buffer is empty. In the case of an explicit receive statement in Occam for example, the buffer can execute a statement: writer? element. In this case the buffer object will wait explicitly until the writer has sent an element. In the case of a communication model without an explicit receive statement methods of the buffer object are executed in the order they are received. This order is dependent on the internal clock of the consumers and the producer and the quality and speed of the connection, which are variable and cannot be predicted. Hence, the former communication type is more deterministic than the latter. However, when comparing synchronous communication to non-blocking communication we find that the latter form introduces even more non-determinism. The extra degree of non-determinism is caused by the fact that computation is continued immediately after sending a message irrespective of whether the message has been accepted. Non-determinism can be reduced by introducing synchronization mechanisms in the concurrency model such that the program can maintain a consistent state. Asynchronous communication decouples the sender from the receiver and therefore behaves better with respect to the autonomous nature of devices in a mobile distributed system.
2) Partial Failures: Also, mobile devices can fail temporarily due to batteries that are drained. Failures are generally a hard problem in the context of distributed computation. In a distributed system a component (network link or device) can fail while the other components in the system are unaffected and continue their computation, hence the name partial failures. In a distributed system a failure generally cannot be detected accurately. Failures are nowadays most often detected based on timeouts. The problem is that timeouts are only an estimation. Latencies of messages can vary based on the load of the network and the machine, such that a message that is considered to be lost because no reply has been received within a certain time interval could still be processed and return the reply too late. It is also possible that a message has been received by a node, but that the link failed just before a reply can be sent. This makes it generally impossible for a sender to determine whether a message that is considered to be lost has actually been received. As a consequence, when such failures are dealt with by sending messages twice it is possible that messages are received multiple times. It is also generally impossible for the sender to determine which component has failed. Either the device or the network link could have failed.

Current mainstream distributed models, such as the ones found in CORBA or other remote method invocations schemes deal with such partial failures by propagating exceptions. However, in mobile networks volatile connections are the common rule rather than the exception. As a consequence programming mobile distributed systems in such models is hard.

A number of conceptual solutions have been developed to deal with failures in distributed systems. The most important ones are (distributed) transactions and replication. A transaction guarantees the atomic execution of a set of actions in the face of failures. Atomic execution means that either all actions are serially executed or none of them at all. Another solution to deal with partial failures is replication. Replication is used to ensure the availability of services in a network by duplicating them on multiple machines in the network such that when a machine or network link fails the service remains available on other nodes. Hence, replicated services try to hide network failures. Other techniques are more application specific. For example, when a device coordinating distributed computations fails a new coordinator could be elected. Although these techniques have proven useful in the context of fixed distributed systems, the protocols associated with these techniques generally do not scale to mobile distributed systems. This is mainly because these protocols typically rely on centralized coordination and expect failures to be rare and of short duration. An example is the 2 phase-commit (2PC) protocol used to support distributed transactions. In the 2-phase commit protocol there is a coordinator that asks all participants in a distributed transaction if they are able to commit the actions associated to the distributed transaction. The participants in the transaction answer “Yes” or “No”. If the coordinator receives a “Yes” from all the participants then it sends a commit instruction to all participants. If any one of the participants answers “No” then the coordinator sends an abort instruction to all participants to all other participants. The result is that the transaction is either committed as a whole or not at all. Note that this scheme only works because a participant that answered “Yes”, cannot change this decision until it receives a “commit” or “abort” message from the coordinator. Hence, if the connection between the coordinator and the participants fails after a number of participants voted “Yes”, then these participants cannot perform any operation that would render its vote invalid. In a fixed distributed system, where failures are exceptional and systems can be closely monitored for failures, such problems can be solved in an acceptable time frame. This is in contrast to the failures encountered in mobile distributed systems that are due to volatile connections. Volatile connections are common and the time to restore a connection can be directly related to the mobility of a user.

IV.OBJECTS VS CONCURRENCY AND DISTRIBUTION

Above we have discussed how concurrency and distribution concepts can generally be addressed in software. We have also discussed the consequences of these choices in the context of mobile distributed systems. Now that we have done this we can turn to a specific paradigm to express concurrency and distribution. The object-oriented programming paradigm provides a good foundation for dealing with distribution and concurrency, because it successfully aligns encapsulated objects with concurrently running distributed software entities. However, there are a number of different approaches how distribution and concurrency issues can be expressed in the paradigm. These approaches are discussed in the following subsections.

A. The Library Approach

Distribution and concurrency primitives are encapsulated and are modeled using the object-based
techniques. Using aggregation and inheritance the primitives can then be integrated in the application. In this approach is that two kinds of objects are used. One kind is used to express the solutions to the issues associated with the concurrency and distribution, while another kind is used to model the domain concepts in the program. Both kinds of objects sometimes need to be mixed to implement the correct solution. An example of this is the Thread class found in many libraries for introducing concurrency in an object oriented language. This example illustrates how concurrency and domain concepts are composed together based on the inheritance relationship. The composition of two different kinds of objects generally results in two problems. A first problem with this approach is that the distinction between domain objects and objects that deal with concurrency and distribution issues is obfuscated. A second problem with this approach is that the library, as in the example above, sometimes enforces a structure onto objects that model domain concepts such that modularizing domain concepts can become impossible. A direct consequence of this is that the extensibility of the different kinds of objects becomes more difficult after they have been composed.

B. The Integrative Approach

The integrative approach aims to align concurrency and distribution concepts with the object paradigm. The integration is achieved by merging some of the concurrency and distribution concepts with the concepts found in the object paradigm. This approach alleviates some of the problems found in the library approach. First, since major concurrency and distribution aspects are merged with concepts of the object paradigm the programmer has to deal with less concepts. This enhances the understandability of the concurrency and distribution aspects of the program. Second, there is less need to manage the concurrency and distribution aspects of a program, provided the object paradigm is aligned intuitively with the concurrency and distribution concepts. The three main dimensions along which concepts can be merged are discussed below.

1) Object and Process: The integration of an object with a process leads to the notion of an active object. The two concepts can be unified because both can be regarded as an encapsulated unit that can communicate with others. An object can have none, one or multiple processes associated with it. An object that does not have any process associated with it is sometimes called a passive object. The number of processes associated with an object gives rise to different types of object-level concurrency:

- Serial or atomic: only one message is computed at a time.
- Quasi-concurrent: multiple object activations within an object can exist at a single point in time. Nevertheless, at most one activation can be executing at a time. The other activations must be suspended at that time.
- Concurrent: multiple unsuspended activations can be present at a single point in time. However, certain restrictions on the concurrency may exist. These restrictions are necessary to maintain a consistent state.
- Fully concurrent: is the same as concurrent objects but without any concurrency restrictions. Fully concurrent object models are functional by nature so that state does not change during a method execution and no inconsistent state can occur.

An important issue with regard to the different types of object-level concurrency is maintaining a consistent state. Quasi-concurrent and concurrent objects are susceptible to race conditions at the level of individual instructions within a method, because concurrent object activations within the same object can result in a non-deterministic interleaving of instructions. On the other hand, serial and fully concurrent object models cannot have race conditions at the level instructions of a method. In the case of serial objects race conditions can still occur at the level of interactions. They give the example of a counter object with set and get methods. Clients want to increment and decrement the counter using these methods. Due to the non-deterministic interleaving of the get and set methods updates can get lost. Suppose the counter is initialized at zero and two clients want to increment the counter by one. Consider the following schedule: both clients request the state of the counter and in both cases the result returned will be zero. Next, both clients update the state of the counter and set it to the result of the get invocation incremented by one. The resulting state of the counter is one. Hence, one counter update can be lost due to the non-deterministic interleaving of messages.

Now that we have discussed the different levels of concurrency that can exist within an object we can turn to how concurrency can be initiated in the object paradigm. There are two approaches objects can be activated: reactive vs. autonomous activation. In the case of reactivity object activation coincides with method invocation. A message is sent to an object and the object is activated by this message. In the
case of autonomy an explicit process is associated with a concurrent object. The object starts running from the moment it is created, with little or no regard to external events. The object paradigm naturally matches better with reactive object activation, but autonomous activation usually gives more fine-grained control over the concurrency issues. For example, autonomous activation offers constructs that allow an object to explicitly receive messages, whereas reactive object models are often based on implicit message acceptance hence, when integrating processes and objects a choice has to be made whether the active object preserves the reactivity principle or whether an autonomous object system is adopted.

2) Object Activation and Synchronization: A second type of integration merges the method invocation and process synchronization concepts. Merging both concepts gives rise to the notion of a synchronized object. When multiple processes are executing in parallel and working on shared resources there is a need to synchronize parts of a program such that it exhibits the correct semantics and prevent that the concurrent accesses lead to an inconsistent state. There are two levels at which synchronization can be integrated with concepts from the object paradigm:

3) Message Passing Level Synchronization: In a sequential object oriented language the sender of a message waits for the receiver to execute the message and return the result of the method invocation. This same mechanism can be used to introduce synchronization between active objects and is also known as synchronous message passing. An active object can send a message to another active object and wait until that object has processed the message and sent back the return value. Message passing forms a natural means to synchronize two concurrently executing objects such that the resulting semantics remains close to sequential semantics. However, in a mobile distributed system, where the latency of messages sent between objects can be high such semantics can harm the autonomous nature of devices. A variant that hides the latency of objects is asynchronous message passing. In this case the sending active object does not wait until the message it sent is actually delivered or even processed. An issue that complicates the use of asynchronous message passing are return values. After all, when an active object does not wait until the called has processed the result it cannot return the result. Typically callbacks are used to process the return values of asynchronous messages, but methods that are used as a callback clutter the code since for each different context in which an asynchronous message is used a callback method needs to be implemented. Another disadvantage of callback methods is that they break the flow of a typical object-oriented program and harm the readability and understandability of the program. To overcome this problem a linguistic abstraction, called futures or promises, have been proposed and implemented in a number of programming languages.

4) Object Level Synchronization: Sometimes more explicit synchronization control is needed that cannot be expressed solely at the message passing level. The necessary degree of control over the synchronization is related to the degree of object passing level:

- Intra-Object synchronization: when multiple object activations within one method can be active at a single point in time there is a need to ensure the consistency of the internal state of the object. Usually, there is a need to specify which methods need to be executed in a mutually exclusive fashion. Note that in a serial active object all methods are mutually exclusive by definition. Although such a serial active object might be considered less expressive, because it restricts the degree of parallelism, it has the benefit that it eliminates inconsistent states that result from concurrent accesses to the internal state of an object.

- Behavioral synchronization: It may be possible that an object, depending on its current state, is temporarily unable to perform methods that are part of its interface. A typical example is a queue that when empty cannot execute an enqueue method invocation until a dequeue method is executed.

- Inter Object synchronization: Sometimes synchronization is necessary between a set of objects to perform a certain task. An example of such a more global synchronization is that of a distributed transaction where a hierarchy of objects are involved to atomically perform tasks. An example of this type of synchronization is a banking application where one account must be credited while a number of other accounts must be debited atomically. More complex synchronization schemes are needed to achieve such synchronization.

The integrative approach minimizes the number of concepts by integrating and unifying concepts of distribution and concurrency. This approach has the advantage that the aspects of distribution and concurrency are more naturally dealt with and are easier to master. However, the integrative approach lacks adaptability and flexibility of the concurrency and distribution concepts offered by the library approach. In other words, the concurrency and
distribution concepts cannot always be adapted to the requirements of the applications.

C. The Reflective Approach

Thus far we have discussed the library and integrative approach. The reflective approach provides a bridge between both approaches. The library approach has the advantage that it allows developers to structure distribution and concurrency into reusable concepts that can be modified thanks to the different extensibility and reusability mechanisms offered by object-oriented techniques. This in effect gives a high degree of flexibility which allows the customization of distribution and concurrency to new contexts. A middle ground between both approaches is the reflective approach. The reflective approach can be regarded as a bridge between the library and integrative approach. The idea is to integrate libraries into the programming language via a meta object protocol (MOP). A MOP allows modifications to the concepts of the object paradigm. In other words, by using the MOP of a language we can unify concurrency and distribution concepts with the language and still have the flexibility offered by the library approach.

CONCLUSION

The two different types of mobile distributed systems and distilled four phenomena that are exhibited by the hardware components used to compose mobile distributed systems have been discussed. Next, we have discussed some software issues that arise when developing distributed systems and considered how the object paradigm can help to structure and develop concurrent and distributed software. There have been a number of proposals for distributed languages that explicitly support open networks. Nevertheless, the current state of the art in distributed languages does not address all the important characteristics that are encountered when developing a nomadic or ad hoc mobile distributed system. On the other hand, middleware approaches offer better, although often incomplete, support to deal with these inherent hardware phenomena of mobile distributed systems. Unfortunately, these approaches do not match well with the object oriented paradigm.

REFERENCES