

Atomic token passing in the context of spontaneous communications

Julien Pauty, Paul Couderc, Michel Banâtre

▶ **To cite this version:**

Julien Pauty, Paul Couderc, Michel Banâtre. Atomic token passing in the context of spontaneous communications. [Research Report] RR-5445, INRIA. 2005, pp.20. inria-00070562

HAL Id: inria-00070562
https://inria.hal.science/inria-00070562

Submitted on 19 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Atomic token passing in the context of spontaneous communications

Julien Pauty — Paul Couderc — Michel Banâtre

No 5445

Janvier 2005

THÈME 1



ISSN 0249-6399



Atomic token passing in the context of spontaneous communications

Julien Pauty , Paul Couderc , Michel Banâtre *

Thème 1 — Réseaux et systèmes
Projet Aces

Rapport de recherche no 5445 — Janvier 2005 — 20 pages

Abstract: With the development of personal wireless mobile devices, direct data exchanges between several devices, known as *spontaneous communications*, will become increasingly important. In such a context, the communication time is limited according to devices' mobility. Despite the limited communication time, applications often require atomic commitment between two mobile devices.

In this paper, we present a protocol to address the problem of atomic token passing from one mobile device to another, in the context of spontaneous communications. The protocol relies on a variant of the two-phase commit protocol. Our approach is based on dynamically adapting the communication time in the two phases of the protocol.

Key-words: ubiquitous computing, atomic commitment, short distance communications, spatial programming

* julien.pauty, paul.couderc, michel.banatre@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex (France)
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Passage de jeton atomique dans le contexte de communications spontanées

Résumé : Avec le développement des terminaux mobiles personnels, les échanges directs de données entre plusieurs terminaux, connus sous le nom de communications spontanées, deviendront de plus en plus importants. Dans un tel contexte, le temps de communication est limité en fonction de la mobilité des terminaux. Malgré ce temps de communication limité, les applications requièrent souvent un mécanisme d'"atomic commitment" .

Dans ce papier, nous présentons un protocole pour traiter le problème du passage atomique d'un jeton d'un terminal mobile à un autre, dans le contexte de communications spontanées. Ce protocole repose sur une variante du "two-phase commit protocol". Notre approche adapte dynamiquement le temps d communication dans les deux phases du protocole.

Mots-clés : informatique diffuse, passage de jeton atomique, communications de courte portée, programmation spatiale

1 Introduction

With new technologies for short-range communications, mobile devices such as mobile phones, PDA or digital cameras, can directly exchange data each other, without the need of a global cellular network. These new communications technologies permit the development of new applications like: business cards exchange; transferring a photo from a digital camera to a mobile phone; gathering information as walking...

Applications synchronized on discoveries of data items are especially important. For example, consider a person who is searching for a flat. Her mobile phone contains a request describing she wants a flat with three rooms, two bathrooms with and a rent bellow one thousand dollars. Now, we suppose that the free flats are electronically tagged. That is, each flat contains a computing device that has capabilities for short-range communications and that embeds the description of the flat. When the person is close to a flat, her mobile phones spontaneously communicates with the flat's device. If the flat's description matches the person's criteria, the application informs her that an interesting flat is close to her.

We call this way of operating *spontaneous communications* [9, 10, 14]. Spontaneous communications enable to create new services that are synchronized on the meetings of two or more physical entities. More generally, spontaneous communications enable to synchronize services on spatial conditions, such as the presence or absence of an object in an area or the meeting of several objects. By their very nature, spontaneous communications are ephemeral: they start on objects meetings and stop as soon as they are out of reach.

Applications relying on spontaneous communications need atomic operations. For example, consider a passenger wanting to take a taxi. The passenger's mobile device contains a stop request for a taxi. When a taxi discovers the request, the request is spontaneously transferred to this taxi, a sound is played inside the taxi's cockpit, the request is deleted from the passenger's device, and finally the taxi stops. Atomic operations are used in situations where there are state changes on multiple devices simultaneously. In the preceding example, the passenger switches from the state "waiting for a taxi" to the state "not waiting". Similarly, the taxi switches from the state "free" to "busy". Atomicity implies that none or both state changes happen. The taxi's state cannot be set to "busy" and the passenger's state left to "waiting for a taxi".

Atomic operations can be implemented with a protocol for atomic commitment. In this paper, we present a protocol to address the problem of atomic commitment in the context of spontaneous communications. This problem is hard, since with spontaneous communications the available communication time is limited. Moreover, spontaneous communications rely on short-range wireless interfaces that can lose packets.

In section 2, we present an overview of spatial programming that is a programming model relying on spontaneous communications. Spatial programming is dedicated to the development of applications involving direct interaction of physical entities, like the taxi or the flat application. In section 3, we introduce the take operation, which is an atomic operation for spatial programming. In section 4, we detail the protocol implementing the take operation. Section 5 presents a mechanism to handle the protocol's failures at the application level. Before concluding, we discuss in section 6 some related works.

2 Spatial programming

In this section we present spatial programming, which is a programming model dedicated to the development of ubiquitous computing [3, 16] applications.

2.1 Principle

Ubiquitous applications are used directly in the physical space, where the user already has and activity. In this context, the user's attention is scarce resource, so the human computer interactions must be limited. Spatial programming limits the human computer interactions by implicitly synchronizing the applications on the user's physical movements. Spatial programming has been used to develop an application to help blind people taking the bus [1], and to develop a museum guide [4].

Spatial programming permits the developer to create applications that are executed in the physical environment over a set of physical objects. These applications are synchronized on spatial interactions between physical objects. Consider for an example a shopping cart. The two possible physical interactions are adding a product inside the shopping cart or withdrawing one. When it detects a new product, the shopping cart adds synchronously the price of the article to the total price. Conversely, when a product withdrawn is detected, the shopping cart synchronously subtracts the price of the product. In this way, spatial interactions implicitly control the application, reducing to a minimum the explicit human computer interactions.

Spatial interactions are reflected at the application level as data exchanges. These data exchanges are done using spontaneous communications.

2.2 Integrating applications in the physical space

A spatial program is mapped on a set of spatial interactions between physical objects, such as objects movements and objects meetings. In this way, the program's execution flow reflects the flow of spatial interactions.

A spatial program uses the physical space as a data store. Each data item fills a defined volume. A physical object can address a data item when the object is inside the corresponding volume. When a physical object is looking for a particular data value, the object can waits either that the data item comes close to him, or moves itself close to this data item.

Spatial programming physically distributes the data in the environment using short-distance wireless communications devices. Each physical object involved in an application embeds an autonomous computing device that is responsible for the data associated to the object. Thus, when an object moves, the data associated to this object moves with it.

2.3 The SPREAD framework

SPREAD is a framework to develop spatial applications. SPREAD represents the data items as tuples and implements a distributed tuple space to store the data; each object embeds

a tuple space that is shared with the neighboring objects. Spatial programming associates each data item to a volume, so, with SPREAD each tuple is also associated to a volume surrounding a physical object. To retrieve a tuple from the tuple space, SPREAD uses a matching mechanism that is similar to the one used in the Linda [6] programming model.

SPREAD proposes several operations:

- read, which reads a tuple and remains blocked until a matching tuple is available;
- check, which checks the presence of a matching tuple and does not remain blocked until a matching tuple is available;
- out, which adds a tuple to the local tuple space;
- drop, which withdraws a tuple from the local tuple space.

When a spatial program executes a read operation, the operation remains blocked until the physical object that executes the operation is inside the volume of the matching tuple. Therefore, when an object's program is blocked on a read operation, the operation is released if the object that embeds the matching tuple comes close to blocked object, or if the blocked object moves itself close to the matching tuple. In this way, programs are implicitly synchronized on objects' movements.

When a spatial program executes a read statement, SPREAD looks for a matching tuple in the local tuple space. If SPREAD does not find any matching tuple, it sends a query to the object in range of communication.

3 The take operation

Previous versions of SPREAD did not propose operation to withdraw a tuple from the tuple space. This kind of operation is needed to perform state changes on two physical objects in one atomic step. We cannot use the drop operation because it works only with tuples stored in the local tuple space. It cannot withdraw a tuple from the tuple space of another object. We call the new operation take.

3.1 Principle

The take operation involves two objects. The first object withdraws a tuple from the second object's tuple space and adds the tuple in its tuple space. Like the read operation, the take operation stays blocked until a matching tuple becomes available.

For example, in the taxi example (figure 1): to search for a passenger the taxi starts a take operation. A passenger who wants a taxi publishes a tuple, which represents a stop request, with an out operation. When a taxi arrives, the take operation "consumes" the stop request of the passenger. That is, the operation withdraws the tuple from the passenger tuple space, adds the tuple to the taxi's tuple space and releases the operation in the taxi's program.

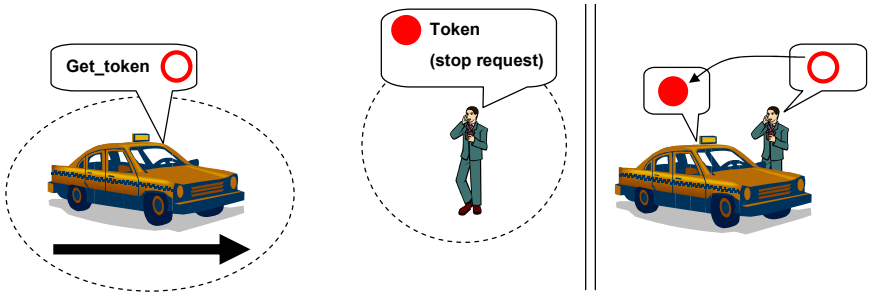


Figure 1: Using the take operation for the taxi application

The take operation is designed to synchronize state changes on the meeting of two physical objects. When the taxi takes the tuple from the passenger's tuple space, the passenger's state and the taxi's state change: the passenger is not waiting for a taxi anymore and the taxi is not free anymore.

The take operation must be atomic. An atomic operation completes as a whole or not complete at all. If an atomic operation uses an intermediate state during its execution, in case of failure, the operation must restore the initial state. In the take operation, atomicity implies that, if the operation fails, the tuple cannot be duplicated nor lost. In the taxi example, we cannot have the take operation released on the taxi and the tuple still in the tuple space of the passenger, which corresponds to a duplicated tuple. Similarly, we cannot have the operation blocked on the taxi and the tuple withdrawn from the passenger's tuple space, which corresponds to a lost tuple.

The take operation must also be *spatially atomic* (figure 2). Spatial atomicity guarantees that the operation is executed during the objects meeting. Ideally, a take operation should be executed instantaneously as the objects meet. Similarly to classical atomicity, spatial atomicity implies that an object cannot be left in an intermediate state. We cannot have the following scenario: two objects meet and a take operation is fired, then, one object leave the area and the operation is paused. Finally, when the objects meet again the operation is finished. In this example, the take operation would be executed over two meetings and the objects are left in an intermediate state. Between the two meetings, they can meet other object and expose this intermediate state, which violates the semantic of atomicity.

Our goal is to propose the take operation in SPREAD in order to ease the programming of applications such as the taxi one.

3.2 The atomicity cannot be guaranteed

The take operation is equivalent to the atomic commitment problem with only two participants. Gray has proved in [7] that, in the presence of message losses, no fixed length protocol exists to solve atomic commitment. Spontaneous communications can lose messages, so, a

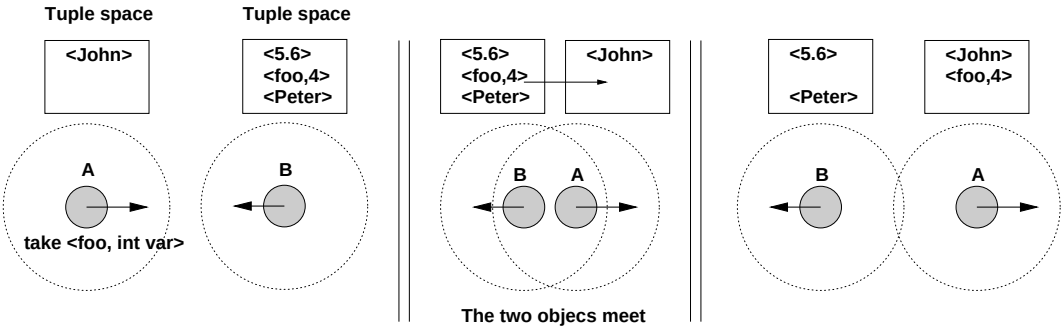


Figure 2: The take operation synchronizes a state change between two tuple spaces belonging to two physical objects A and B

protocol that guarantees that the take operation is atomic cannot be fixed length. It would require potentially an infinite number of messages.

By their nature, spontaneous communications have a limited duration. Therefore, over spontaneous communications, only fixed length protocols can be used. Consequently, the take operation cannot guarantee strict atomicity over spontaneous communications.

A take operation that is not atomic is a failure. The errors that cause these failures are message losses, which are caused by network errors such as shadowing problems or electromagnetic perturbations. In the context of spontaneous communications, packets can also be lost because the objects are too far to communicate.

Over spontaneous communications, we cannot guarantee that there will be no failure, but we can try to limit them. In this paper, we present a protocol that limits the number of failures. We also present how to handle the residual failures at the application level.

3.3 Modeling the take operation as a token passing operation

The semantic of the take operation is the following: an object A is waiting for a tuple ; when the object A meets an object B, which has a matching tuple in its tuple space, the object B withdraws the tuple from its tuple space and sends it to the object A.

The problem of the atomicity of the take operation is equivalent to the problem of passing a token atomically from one physical object to another: an object A is waiting for a token to follow its execution; when the object A meets an object B, which has a token, the object B gives the token to the object A, which can continue its execution. In the rest of this paper, for sake of clarity, we rely only on this model.

4 Protocol of the token passing operation

In this section, we present the protocol of the token passing operation. The protocol is based on the two-phase commit protocol. If the protocol can finish then it guarantees that the operation is atomic. If messages are lost during the protocol, then the operation may not be atomic.

We distinguish two categories of faults that cause message losses: (i) network errors; (ii) the objects are too far to communicate. The first category of fault can be handheld by a standard retry mechanism. The second category of fault is specific to spontaneous communications. In this case, the protocol may fail because the objects do not have enough communication time. To prevent this situation, the protocol uses a geometric constraint to guarantee that the objects have a minimum communication time.

For applications where no failure must happen, if the objects do not have enough communication time, the protocol can fall back to a global network to extend the communication time and terminate the operation.

4.1 The two-phase commit protocol

The two-phase commit protocol has been proposed to solve the problem of making atomic the commit phase of a transaction in a database. The problem of passing a token between to physical objects is a simpler version of the atomic commitment problem. This problem is solvable by relaxing the restriction that the protocol has a fixed length. Therefore, the 2PC protocol may require an infinite number of messages. The 2PC is not a fixed length protocol.

The 2PC protocol relies on a commit coordinator. During the first phase, the coordinator asks each participant in the transaction to vote. If a participant votes yes then it is "prepared" to commit. That is, it is in a state from where it can rollback to its initial state. If all participants vote yes, then the first phase is finished and the commit coordinator records a commit record. Once the commit record is saved, the second phase begins and the commit coordinator broadcasts a message to all participants to tell them to validate the commit. If one participant votes no during the first phase, then the commit coordinator broadcasts a message to all participants to abort the transaction.

The 2PC protocol tolerates message losses and participant crashes. If an error happens during the first phase then the protocol can make some retries (possibly an infinite number) and can eventually abort the transaction. If an error happens during the second phase, the commit coordinator must retry to send the commit confirmation until every participant has acknowledged the commit confirmation. Therefore, the communication time needed to terminate the 2PC commit protocol may be infinite. Consequently, over spontaneous communications, the common 2PC protocol cannot be used. It must be adapted.

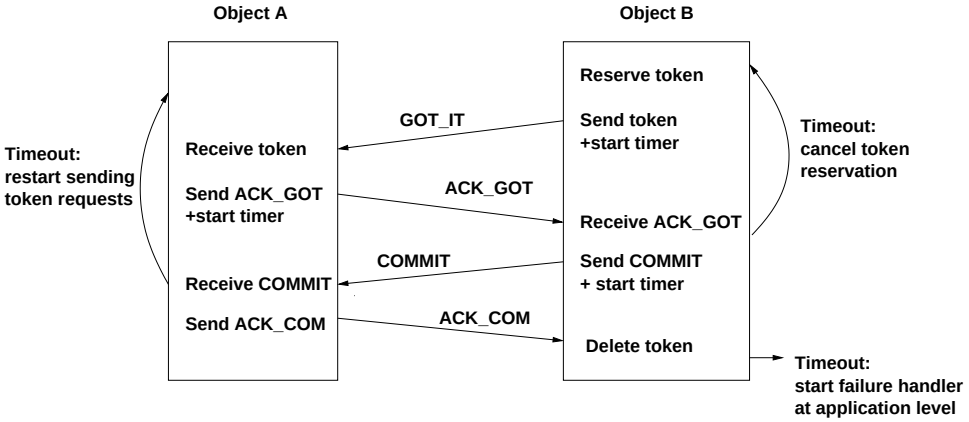


Figure 3: Protocol of the token passing operation

4.2 An adaptation of the 2PC protocol for spontaneous communications

In this section, we present an adaptation of the 2PC protocol to implement the atomic token passing operation over spontaneous communications. The token passing operation involves two objects: the Tok object that owns the token and the Req object that makes the token request.

4.2.1 Principle

The two participants of the protocol are the two physical objects involved into the token passing operation. The participant who owns the token is also the commit coordinator. During the first phase, the commit coordinator reserves the token so that it becomes unavailable for other token passing operations. Then, the commit coordinator sends the token to the second participant. During the second phase, the commit coordinator sends a commit message to the second participant and erases the token from its memory. When the second participant receives the commit message, it releases the token passing operation.

The protocol for the token passing operation has four messages (figure 3): `GOT_IT`, `ACK_GOT`, `COMMIT` and `ACK_COMM`. The object Req requests a token by regularly broadcasting requests for a token. When the object Tok receives the request, the token passing operation starts: Tok reserves the token for this operation and sends the token to the object Req inside a `GOT_IT` message. Then, the object Req acknowledges the `GOT_IT` message by sending a `ACK_GOT` message. When Tok receives `ACK_GOT` the first phase is finished. During the second phase, Tok sends a `COMMIT` message to Req, and Req acknowledges with a `ACK_COMM` message. When Req receives `COMMIT`, it releases the token passing operation. When Tok receives `ACK_COMM`, it erases the token from its memory.

4.2.2 Protocol failures

The protocol of the token passing operation is executed over spontaneous communications, which are inherently unreliable, so messages can be lost during the protocol.

Each of the four messages can be lost during the protocol. During the first phase of the protocol, if the `GOT_IT` message or the `ACK_GOT` message are lost the operation aborts and the system stays in a consistent state. There is no failure. If the `GOT_IT` message is lost, then, after a timeout, Req restarts sending token requests. If the `GOT_IT` message is lost, Tok cannot receive the `ACK_GOT` message. After a given timeout, Tok aborts the token passing operation and frees the token. Similarly, if the `ACK_GOT` message is lost, then Tok aborts the operation and frees the token.

During the second phase of the protocol, if the object Tok does not receive the `ACK_COMM` message, then it means that the `COMMIT` message or the `ACK_COMM` message has been lost. If Tok does not received the `COMMIT` message after a given time, it aborts the operation and restarts sending token requests; the operation failed.

When the operation fails the system can be in two different inconsistent states. If the `COMMIT` message has been lost, then the token passing operation is still blocked on the object Req and it has restarted sending token requests. On the object Tok, the token is still reserved. If the `ACK_COMM` has been lost, the object Req has received the token and the token passing operation has been released. On the object Tok, the token is still reserved.

4.3 Reducing the number of failure

Failures happen when messages are lost in the second phase of the protocol. To improve the reliability of a protocol a common solution is to add a retry mechanism. For our protocol, we can add some retries for the `COMMIT` message.

A retry mechanism must be used with cautions, because it can violate the property of spatial atomicity. By adding retries to the protocol, we increase the communication time of the protocol, which in turn extends the area in which the objects can move and complete the protocol. In fact, if we allow an infinite number of retry, then the communication time can be infinite. If the number of retries is too large, we can have the situation where an object leaves and then re-enters in the communication range of another object. This kind of situation violates the spatial atomicity of the operation.

The critical parameters of a retry mechanism are the number of retries and the delay between them. This number is determined according to the available communication time, which is dependent on the speed, the location and the trajectory of the objects. Thus, the number of retries is difficult to determine. A solution has been proposed in [15] for a similar problem, which addresses the automatic discovery of mobile devices using presence messages.

In this section, we propose another approach to limit the number of failed operations. This approach uses a geometric constraint that prevents an operation to start when the distance between the two objects is too big.

4.3.1 Principle

If the physical objects involved in the operation do not have enough communication time to terminate the protocol, message losses may happen during the execution of the token passing operation, leading to failed operations. Using a wireless network, the nodes have a communication range that can be approximated by a sphere. The communication time between two physical objects depends on their relative location and speed. If the communication time is too short, then the protocol may not have enough time to terminate, meaning that either the `COMMIT` message or the `ACK_COMM` message can be lost.

To reduce the number of failures, we propose to define a geometric constraint that guarantees a minimum communication time when an operation starts. This geometric constraint is composed of a threshold distance. A token passing operation can start (figure 4), only when the distance between the two objects is bellow this threshold. For example, we consider two objects with a communication range of 300 meters, a speed of 13 meter per second. The shortest communication time happens when the objects move in opposite directions. In this case, if the threshold is set to 50 meters, then the length of the shortest path is 250 meters. Therefore minimum available communication time is $(250 - 50)/(13 * 2) = 7.7s$. In practice, the distance between two objects can be obtained with a location technology like the GPS or by measuring the strength of the radio signal used to communicate.

4.3.2 Evaluation of the geometric constraint

For practical reasons, we evaluate the geometric constraint with a simulation. We evaluate the impact of this constraint using the NS simulator with the wireless extension. During the simulations, the physical objects communicate with 802.11b network interfaces (WiFi). The simulation is set up for an outdoor urban environment.

First scenario: approaching objects For our first evaluation, we consider a scenario similar to the taxi application's scenario. We have a mobile object (the taxi) which is searching for a token and a static object (the passenger) who owns the token. The mobile object's speed is 50 kilometers per hour. A simulation is composed of a series of runs. At the beginning of each run, the mobile object is outside the communication range of the static object and moves toward the static object (figure 5 left). A run terminates when the token passing operation is finished. Only one token passing operation is executed per run. For this first simulation there is no retries in the protocol.

During the first simulation, there is no geometric constraint. This means that the token passing operation can start as soon as the objects can communicate. Then we run again the simulation with a constraint set to 290 meters. This time, the token passing operations can start only when the distance between the two objects is bellow 290 meters. The simulation is executed several times with the constraint ranging from 290 meters to 10 meters.

During each simulation, the number of failure is counted. A failure happens when the `COMMIT` message or the `ACK_COMM` message is lost. For each simulation, we have n_s successful token passing operations and n_f failed token passing operations. We want to get the failure

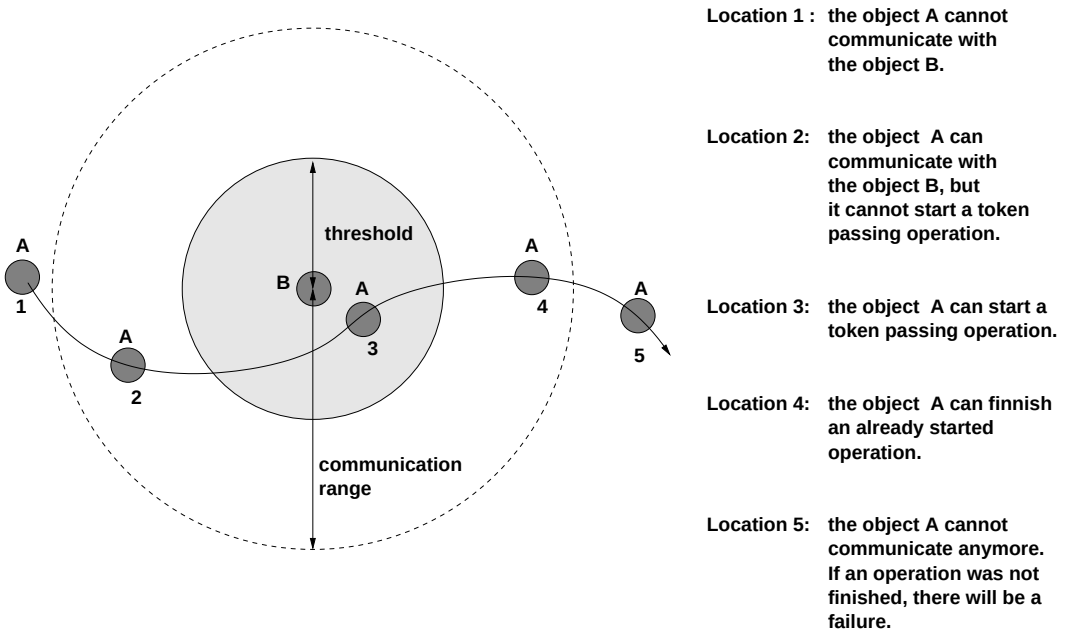


Figure 4: A geometric constrain guarantees a minimum communication time between the object A and the object B

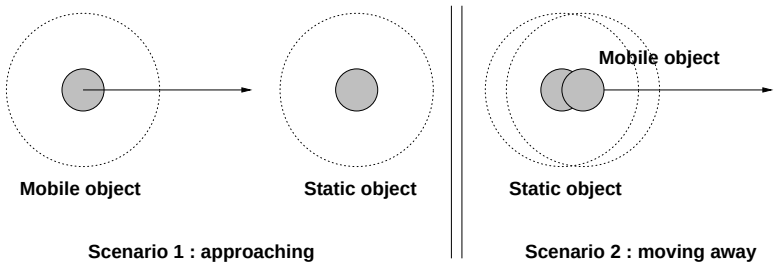


Figure 5: The two scenarios used in the simulations

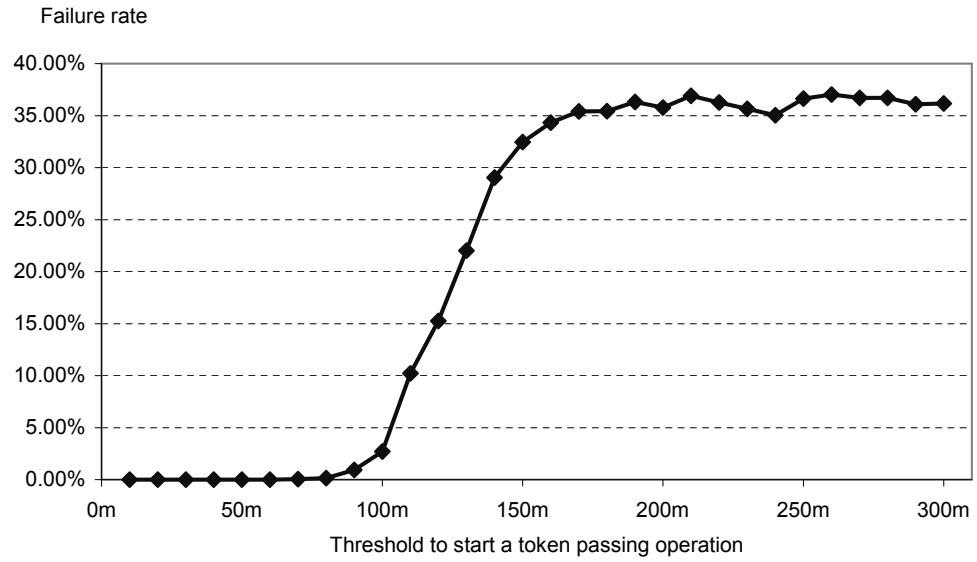


Figure 6: Failure rate with no retries. Scenario 1

rate r that represents the quantity of failures according to total number of tried operations: $r_f = n_s/(n_f + n_s)$. Results are presented on figure 6. On the curve, the point for a threshold of 300 meters represents the failure rate without any constraint, since the theoretical communication range of WiFi is 300 meters.

Analysis For this simulation, we see that with a threshold of 60 meters there is no failed operation. This is an important result because in this scenario there is no retry in the protocol. Therefore, by setting the threshold to 60 meters, for this simulation, we can guarantee that the operation is spatially atomic without any retry. By setting the threshold to 10 meters the operations are also spatially atomic. However, with a small threshold, the opportunities of take operations are rare. Thus for best efficiency, we must chose the largest threshold, inside the set of thresholds that guarantee a null failure rate. In addition, the absence of retry saves energy and bandwidth.

By setting the threshold between 300 and 170 meters the failure rate ranges from 34 to 35 percents. Since a threshold of 300 is equivalent to no threshold at all, this means that the constraint has no effect on the failure rate for large threshold. In fact, even if maximum theoretical range is 300 meter, in practice objects hardly communicate beyond 170 meters. For that reason, the constraint starts to affect the failure rate only when the threshold is below 160 meters.

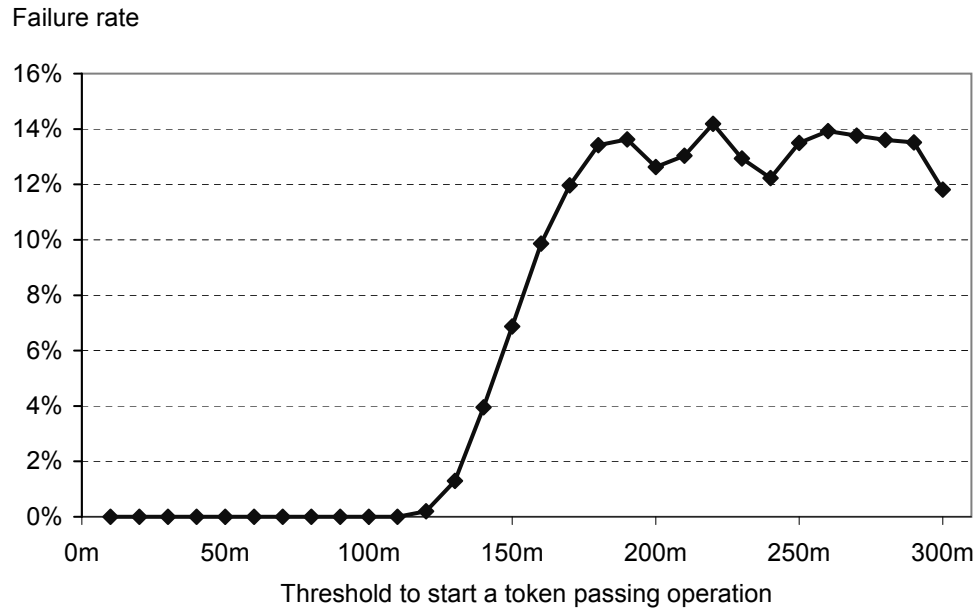


Figure 7: Failure rate with two retries. Scenario 1

Adding retries We keep the first scenario, but we add retries in the second phase of the protocol. The maximum number of retries is set to two, meaning that we have a maximum of three `COMMIT` messages sent per operation. We keep the number or retries low to limit the risks that the spatial atomicity is violated. The retries extends the area where the protocol can terminate. In this case we have two retries, which increase the communication time of one second, implying a circle with a radius of 14 meters. We run the same set of simulations. Results are on figure 7 An interesting result with this simulation is that with retries the failure rate is null with a threshold below 100 meters, which is better than without retries. In addition, when the threshold is comprised between 300 and 170 meters the failure rate is lower and ranges from 12 to 14 percents.

Second scenario: moving away objects The first simulation is a favorable case because the communication time is maximized since the objects are approaching. Now, we want to determine which constraint provides the lowest failure rate when the objects are moving away, which gives the shortest communication time.

In this scenario, we still have one static object that owns the token and one mobile object that wants the token. A simulation is still composed of a series of runs. At the beginning of a run, the objects are at the same location and the mobile object starts moving away from the static object (figure 5 right). During the first series of simulation runs, the token passing

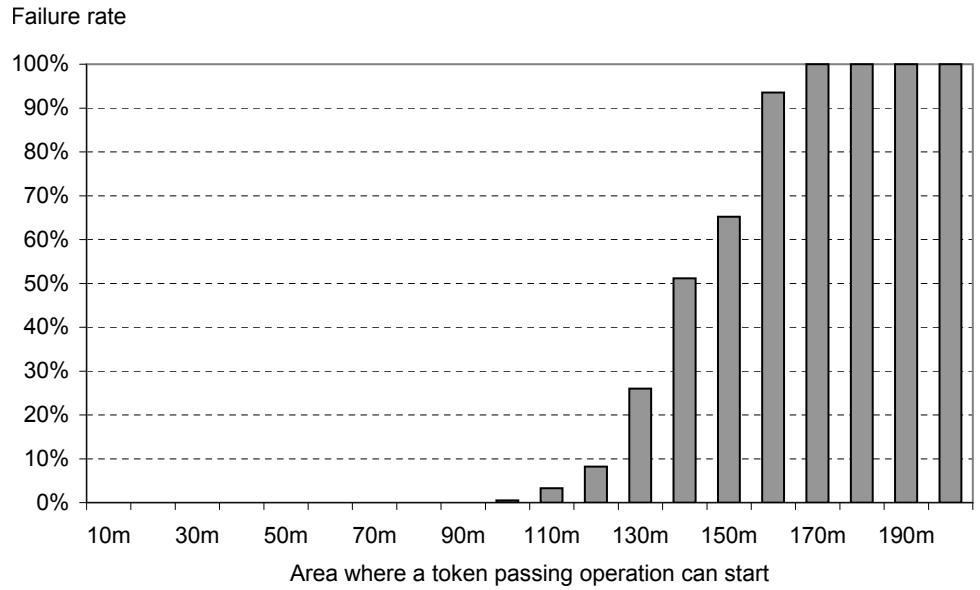


Figure 8: Failure rate according to the area where the operations are started. With two retries. Scenario 2

operations start when the distance between the two objects is below 10 meters. During the second series of simulation runs, the operations can start only between 10 and 20 meters, and so on. For these simulations, we keep two retries for the `COMMIT` message. Results are on figure 8.

This simulation shows that if the operation is started between zero and 90 meters there is no failure. Between 160 and 210 meters, we have a failure rate of 100%. However, in this area we have very few operations that happen, less than 0.5%. In this area, the network is so unreliable that it is very difficult to start an operation. Consequently, it is even more difficult to complete a started operation.

4.4 Restoring consistency with a global network

To restore consistency, the application can fall back to a network with a global coverage, like the GPRS network, to restore the consistency. This could be useful for applications that cannot tolerate any inconsistency.

Errors happen because the physical objects do not have enough communication time to terminate the protocol of the token passing operation. That is, the physical objects cannot communicate anymore because they are too far. To terminate the protocol, the communication time must be extended. The communication time depends on the objects'

speed and objects' communication range. Altering the object's speed is not possible as we consider autonomous objects. Thus, the only possibility to increase the communication time is to extend the communication range of the objects.

When the short-range communications fail, using a global network guarantees the eventual termination of the take operation. However, it may violate the semantic of the operation, since, as the global network is used, the meeting between the objects is already finished.

Extending the network range by using a global cellular network would imply a higher communication cost, since global networks are not free, unlike spontaneous communication relying on an ad-hoc WiFi mode. Moreover, global communications, like GPRS communications, consume more energy than spontaneous communications.

We could also envision a network interface with a variable communication range. The physical object could choose the communication range, a greater range implying a greater energy consumption.

5 Failure handling at application level

In the preceding section, we have presented how a geometric constraint can reduce the number of failed token passing operations. However, this constraint does not guarantee that there will not be any failure. Since failures lead to inconsistent states, we have to deal with these issues. In particular, we have to try to restore the consistency of the system.

In this section, we still consider the Tok object that owns the token and the Req object that makes the token request.

5.1 Failure detection

In the presence of message losses, the system can enter in two different inconsistent states: (i) the object Req did not get the token and the token is still reserved on the object Tok (ii) the object Req received the token and the token is still reserved in the object Tok. The first case can be seen has a token lost and the second case as a token duplication. However, in the first case the token is not truly lost, because the object Tok still has the token in its memory. Similarly, in the second case the token not truly duplicated, because the token is reserved on the object Tok and therefore not available for another token passing operation.

Only the physical object Tok is able to detect that the system is in an inconsistent state. In each inconsistent state, the physical object Tok is blocked in the second phase of the in protocol with the token still reserved. After a given number of retries, Tok declares that its local state is inconsistent and thus detect that the system is in an inconsistent state. Req cannot detect the inconsistency because, in each global inconsistent state, its local state is consistent: it is either waiting for a token or it has released the token passing operation.

5.2 Restoring consistency

Since this is the object Tok that detects the inconsistent state, which is caused by a failed operation, it is the object Tok that must restore the consistency. When the object Tok detects a failure, it knows that some object wanted to take one of its token and that the operation has failed. Thus, it must decide if the token must be freed or deleted. To take its decision, the object Tok needs information describing the situation.

5.2.1 Information needed to restore consistency

At the protocol level, the only available information is that the `COMMIT` message did not arrive, which means that either the `COMMIT` message or the `ACK_COMM` message has been lost. This information is not sufficient to restore consistency. If the message `COMMIT` has been lost, then the object Req has restarted sending token requests, so the object Tok should free its token. If the `ACK_COMM` has been lost, then the object Req has got the token, so the object Tok should delete its token. The problem is that Tok cannot differentiate between the first and the second case.

The information at the protocol level is not sufficient to restore the consistency, but the object Tok can get further information at the application level. Actually, a token is a tuple. That is, the application is able to distinguish one token from another. When an operation is started, the object Tok reserves a corresponding token for this operation. According to the nature of the token, to restore consistency, the object Tok can take a decision on a case-by-case basis. For each token, the application developer must plan how to restore consistency when the token passing fails. If it is not possible to plan how to automatically restore consistency at the application level, the help or the notification of the user may be required

To restore consistency SPREAD offers a callback mechanism that associates a different callback function to each token. SPREAD calls this function when the token is involved in a failed operation.

5.2.2 Example

We consider a taxi that is requesting a token and a passenger that creates a token to say he wants a taxi. When the taxi gets a token the taxi stops to take the passenger. If the token passing operation fails, the taxi was probably too far from the passenger.

The taxi cannot detect this failure. If the `COMMIT` has been lost, the taxi has aborted the operation and has started again requesting a token. If the `ACK_COMM` message has been lost, the taxi has got the token and stops. If the taxi driver sees nobody waiting for a taxi, it decides to leave and launch a new the token request. If the taxi driver sees the passenger then the passenger comes into the taxi and the taxi leaves.

From the passenger point of view, when the failure happens, if the taxi is not visible then the token should be unreserved to request another taxi. If the taxi is visible and has stop, then this is the `ACK_COMM` message that has been lost and the token must be deleted.

To restore consistency on the passenger side, we must know if the taxi is visible, so only the passenger can take a decision. Consequently, the application has to report to the passenger there have been a problem and ask him if he still wants to request a taxi or if a taxi has stop for him. Here the callback function just asks a question to the passenger.

For applications where losing a token is not critical, the developer can decide to systematically erase the token involved in failed token passing operation. Conversely, for other kinds of token it may be better to free the tokens involved in failed operations. In this later case, we may have duplicated tokens.

6 Related works

A lot of work has been done around atomic commitment [2, 12]. These works assume that the communication network is reliable, meaning that the messages eventually arrive. Unfortunately, in our context, we cannot consider that the messages eventually arrive, since the physical objects may simply be too far to communicate.

Atomic commitment has been studied also in the area of mobile computing. Dunham has proposed the Kangaroo model [5], as a model for transactions processing for mobile users. In the Kangaroo model, each mobile node is connected to a base station that acts as a gateway to the fixed network. In our scenario, there is no network at all and the mobile nodes communicate only with spontaneous communications, except when a global network is used to restore the consistency. Moreover, the Kangaroo model is not a spatial model; a transaction is not associated to a precise location. This transaction model hides the problems due to nodes' mobility, unlike spatial programming where the mobility drives the applications.

Our work shares similarities with systems handling inconsistencies at the application level, like the Bayou system [13]. The Bayou system offers a weak consistency model for replicated data. With the Bayou system, a data item may have different values on different servers. It is up to the application programmer to write the code to restore consistency. Like our protocol, with the Bayou system the user may be prompted to restore the consistency. The user is prompted in cases where his knowledge is needed to restore consistency.

Lime [11] is a system that permits to mobile nodes to coordinate each other via a distributed tuple space. Lime is not a spatial programming model, the data items does not fill a volume. The approach of Lime hides the impact of the mobility to the applications. Lime tries to always give a consistent and up to date view of the tuple space to the processes. SPREAD takes an opposite approach. The objects movements directly drive the application, so the application developer directly maps its programs to these movements.

7 Conclusion

In this article, we have presented a protocol to address the problem of atomic token passing in the context of spontaneous communications. Our protocol limits the number of failures

by enabling a token passing operation to start only when a minimum communication time is available. If a failure happens and if a global network is available, the consistency can be restored temporarily using the global network, actually extending the communication time. If a global network is not available, applications may have to restore the consistency in an ad-hoc manner.

Spatial applications, which are synchronized on physical objects' meetings, need atomic token passing. In the taxi application, when the taxi meets the passenger it gets the token and stops. More generally, atomic token passing is needed when several objects want to interact over spontaneous communications.

In our future works, we will continue to evaluate the protocol of the token passing operation, especially with more scenarios of spatial applications. We will also implement the protocol on various architectures, like motes [8] and PDA, to do evaluations in real life conditions. Finally, over these architectures, we will use the take operation to implement new spatial applications.

References

[1] M. Banâtre, P. Couderc, J. Pauty, and M. Becus. Ubibus: Ubiquitous Computing to Help Blind People in Public Transport. In *Mobile HCI 2004*, pages 310–314, 2004.

[2] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225–267, 1996.

[3] G. Chen and D. Kotz. A Survey of Context-Aware Mobile Computing Research. Technical Report TR2000-381, Dept. of Computer Science, Dartmouth College, November 2000.

[4] P. Couderc and M. Banâtre. Spreading the web. In Springer, editor, *Personnal Wireless Communications (PWC'03)*, pages 375–384, 2003.

[5] M. H. Dunham, A. Helal, and S. Balakrishnan. A Mobile Transaction Model That Captures Both the Data and Movement Behavior. *Mobile Networks and Applications*, 2(2):149–162, 1997.

[6] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, 1985.

[7] J. Gray. Notes on Data Base Operating Systems. In *Operating Systems, An Advanced Course*, pages 393–481. Springer-Verlag, 1978.

[8] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System Architecture Directions for Networked Sensors. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS'00)*, pages 93–104, 2000.

[9] G. Kortuem, J. Schneider, D. Preuitt, T. G. C. Thompson, S. Fickas, and Z. Segall. When peer-to-peer comes face-to-face: Collaborative peer-to-peer computing in mobile ad hoc networks. In *P2P '01: Proceedings of the First International Conference on Peer-to-Peer Computing (P2P'01)*, page 75. IEEE Computer Society, 2001.

[10] Gerd Kortuem, Zary Segall, and Thaddeus G. Cowan Thompson. Close encounters: Supporting mobile collaboration through interchange of user profiles. In *HUC '99: Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing*, pages 171–185. Springer-Verlag, 1999.

[11] G. P. Picco, A. L. Murphy, and G.-C. Roman. LIME : Linda Meets Mobility. In *International Conference on Software Engineering (ICSE'99)*, pages 368–377, 1999.

[12] M. Raynal. Revisiting the Non-Blocking Atomic Commitment Problem in Distributed Systems. In *IPPS IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, pages 116–133, April 1997.

[13] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 172–182. ACM Press, 1995.

[14] D. Touzet, J-M. Menaud, M. Banâtre, P. Couderc, and F. Weis. SIDE Surfer: a Spontaneous Information Discovery and Exchange System. In *Proceedings of the Second International Workshop on Ubiquitous Computing and Communications (WUCC'2001)*, Barcelona, Spain, September 2001.

[15] A. Troël, M. Banâtre, and F. Weis. Automatic neighborhood discovery between wireless mobile appliances. Technical report, IRISA, 2004.

[16] M. Weiser. Some Computer Science Issues in Ubiquitous Computing. *Communications of the ACM, Back to the Real World, Special issue on Computer Augmented Environments*, 36(7):75–84, 1993.



Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399