Programming Mobile Context-aware Applications with TOTAM

Elisa Gonzalez Boix[1,*], Christophe Scholliers, Wolfgang De Meuter, Theo D'Hondt

Software Languages Lab
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, Belgium

Abstract

In tuple space approaches to context-aware mobile systems, the notion of context is defined by the presence or absence of certain tuples in the tuple space. Existing approaches define such presence either by collocation of devices holding the tuples or by replication of tuples across all devices. We show that both approaches can lead to an erroneous perception of context. Collocation ties the perception of context to network connectivity which does not always yield the expected result. Tuple replication can cause that a certain context is perceived even if the device has left the context a long time ago. We propose a tuple space approach in which tuples themselves carry a predicate that determines whether they are in the right context or not. We present a practical API for our approach and show its use by means of the implementation of various mobile applications. Benchmarks show that our approach can lead to a significant increase in performance compared to other approaches.

Keywords: tuple spaces, programming abstractions, context-awareness, mobile peer-to-peer systems

1. Introduction

A growing body of research in pervasive computing deals with coordination in *mobile ad hoc networks*. Such networks are composed of mobile devices which spontaneously interact with other devices within communication range as they move about. This network topology is often used to convey context information to collocated devices [1, 2, 3, 4, 5]. Moreover, such context information can be used to optimize application behaviour given the scare resources of mobile devices [2]. In this paper, we focus on distributed programming abstractions to ease the development of context-aware applications deployed in a mobile environment.

Developing these applications is complicated because of two discriminating properties inherent to mobile ad hoc networks [6]: nodes in the network only have inter-

*Correspondence to: Elisa Gonzalez Boix; Software Language Lab, Vrije Universiteit Brussel, Pleinlaan, 2 1050 Brussel, Belgium; e-mail: egonzale@vub.ac.be; tel: +32 2 629 3956; fax: +32 2 629 3525

mittent connectivity (due to the limited communication range of wireless technology combined with the mobility of the devices) and applications need to discover and collaborate without relying on a centralized coordination facility. Decoupled coordination models such as tuple spaces provide an appropriate paradigm for dealing with those properties [2]. Several adaptations of tuple spaces have been specially developed for the mobile environment (including LIME [1], L2imbo [3], EgoSpaces [4], and TOTA [5]). In those systems, processes communicate by reading from and writing tuples to collocated devices in the environment. Context information in such systems is thus represented by the ability to *read* certain tuples from the environment. In this paper we argue that this representation is inappropriate and can even lead to an erroneous perception of context. The main reason for this is that the ability to read a tuple from the environment does not give any guarantees that the context information carried by the tuple is appropriate for the reader. This forces programmers to manually verify that a tuple is valid for the application's context situation *after* the tuple is read.

To support the development of mobile context-aware applications, we propose a novel tuple space approach called TOTAM ("Tuples On The Ambient") which decouples the concept of tuple perception from tuple reception. A TOTAM tuple can contain an associated predicated called a *context rule* that describes the runtime conditions under which it is visible to applications. Only when a tuple's context rule can be satisfied by the context of the receiving application, the tuple can be perceived by the application. Applications can also be notified when the tuple can no longer be perceived. The core contribution of this work lies in proposing a tuple space model designed for a mobile environment which introduces a general programming concept under the form of a context rule to support development of mobile context-aware applications. Our contribution is validated by (1) a prototype implementation, (2) demonstrating the applicability of our model by using it in non-trivial context-aware mobile applications, (3) using our prototype implementation for teaching students, (4) providing an operational semantics for our model, (5) benchmarking our prototype implementation.

TOTAM has been employed in a realistic experiment with the Brussels public transport company. In addition, students have used our tuple space based approach for the development of various mobile peer-to-peer applications. These experiences have led us to significantly improve previous iterations of our tuple space model [7] in two fundamental ways. First, we have developed a *leasing model* which allows the underlying system to reclaim tuples in the face of partial failures. Second, we have revisited the *propagation mechanism* included in TOTAM which allows developers to *scope the distribution of tuples* in the network. In this paper, we report on the various extensions that we have applied over our original tuple space abstractions and how these extensions ease the development of mobile context-aware applications.

2. Motivation

Tuple spaces were first introduced in the coordination language Linda [8]. Recently they have shown to provide a suitable coordination model for the mobile environment [2]. In the tuple space model, processes communicate by means of a globally shared virtual data structure (called a tuple space) by reading and writing tuples. A tuple is an ordered group of values (called the tuple content) and has an identifier (called

the type name). Processes can post and read tuples using three basic operations: `out` to insert a tuple into the tuple space, `in` to remove a tuple from the tuple space and `rd` to check if a tuple is present in the tuple space (without removing it). Tuples are anonymous and are extracted from the tuple space by means of pattern matching on the tuple content.

In order to describe the main motivation behind TOTAM, we introduce a simple yet representative scenario and show the limitations of existing tuple space approaches. Consider a company building where each room is equipped with devices that act as *context providers* of different kinds of information. For example, information to help visitors to orient themselves in the building or information about the meeting schedule in a certain room. Employees and visitors are equipped with mobile devices which they use to plan meetings or to find their way through the building. Since each room is equipped with a context provider, a user located in one room will receive context information from a range of context providers. Only part of this context information which is broadcasted in the ambient is *valid* for the current context of the user.

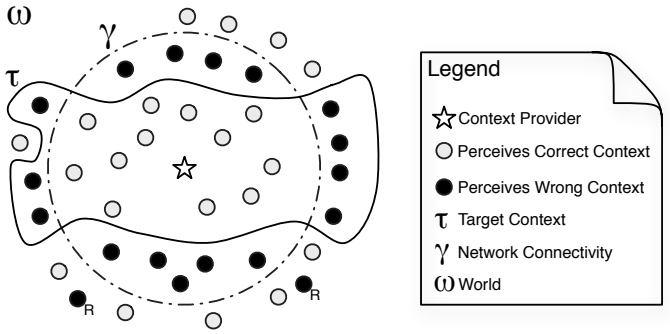

Figure 1: Context perception in existing tuple space approaches

Figure 1 illustrates the scenario where ω represents the company building, τ a meeting room in the building, and γ the communication range of a device located in the meeting room. The star denotes a tuple space (acting as the context provider) which injects tuples into the ambient, i.e. all devices (depicted as dots) $\in \gamma$. Those tuples are aimed to be perceived by devices in the meeting room, i.e. in the target area τ. This device injects a tuple in the ambient to signal receivers that they are currently in the meeting room. Note that location is just one example of context, τ could involve more complex constraints, e.g. being located in the meeting room while there is a meeting.

We now give an overview of how context is perceived in the two most prominent tuple space models. A first group of tuple space models, follows a *federated* tuple space model [1] in which the visibility of the tuples (and thus context perception) directly depends on collocation of devices holding these tuples. In this model, the perceived context of a device is equivalent to being in range of γ. The context delivery solely based on γ makes two groups of devices to perceive wrong context information (depicted as black dots). The first group consists of devices contained in the set $\gamma \setminus \tau$. In

our example, these are all devices within communication range of the context provider but outside the meeting room. These devices will perceive to be in the meeting room while they are actually not. The second group consists of devices contained in the set $\tau \setminus \gamma$. In our example, these are all devices out of communication range of the context provider (possibly due to an intermittent disconnection) but in the meeting room. These devices will perceive not to be in the meeting room while they actually are.

Other tuple space systems have adopted a *replication* model where tuples are replicated amongst collocated devices in order to increase data availability in the face of intermittent connectivity [5, 9]. In replication-based models, devices in $\tau \setminus \gamma$ will not perceive wrong context information. However, in these systems tuple perception is equivalent to have been once within reach of γ, possibly in the past. This means that devices which have been connected to the context provider once (τ) and are currently in $\omega \setminus \tau$ (depicted as black dots with a R) will perceive to be in the meeting room even though they are no longer there.

2.1. Summary

Using current tuple space approaches the context perception is correct in certain cases (the white dots) but, in many cases it is wrong (black dots). There are three main reasons for these erroneous context perceptions. First, there is a connectivity-context mismatch making context sharing based *solely* on connectivity unsuitable for the development of context-aware applications deployed in a mobile setting. Second, the observed context is affected by intermittent connectivity: temporal disconnections with the context provider result in an erroneous context perception. Third, when using replication-based models to deal with intermittent connectivity, a permanent disconnection leads devices to perceive a certain context forever.

In order to solve these issues, programmers are forced to manually verify that every tuple (and thus context information) is valid for the application' context. More concretely, programmers have to manually determine tuple perception at the application level after a tuple is read from the tuple space. Manually determining the applications context and adapt accordingly leads to context-related conditionals (`if` statements) being scattered all over the program [10], hindering modularity. Additionally, the content of the tuples have to be polluted with meta data in order to infer tuple perception at application level, decreasing reusability of tuples. For example, a `Room` tuple indicating that a person is currently located in the meeting room should also contain the location information. Finally, programmers need to write application level code that deals with context-awareness in order to compensate for the lack of expressiveness of underlying model.

As the complexity of context-aware applications increases, manually computing tuple perception can no longer be solved using ad hoc solutions. Instead, the coordination model should be augmented with abstractions for context-awareness that allow developers to describe tuple perception in the coordination model itself. In this work, we introduce TOTAM, a novel approach that keeps the simplicity of the tuple space model where interactions and context information are defined by means of tuples, while allowing tuples themselves to determine the context in which a receiving application should be in order to perceive a tuple.

3. The TOTAM Approach

TOTAM is a novel tuple space-based programming model for mobile context-aware applications. It adopts concepts from both federated and replication-based tuple spaces, and extends them in order to overcome the aforementioned tuple perception issues. The TOTAM system introduces the notion of a *context rule* prescribing when a tuple should be perceived by the application, and a *rule engine* to infer when a tuple is perceivable and when it is not. TOTAM has a scoping mechanism (inspired by TOTA [5]) and a leasing model that allows developers to deal with the hardware characteristics of a mobile setting. In the remainder of this section, we describe the most important features of our model.

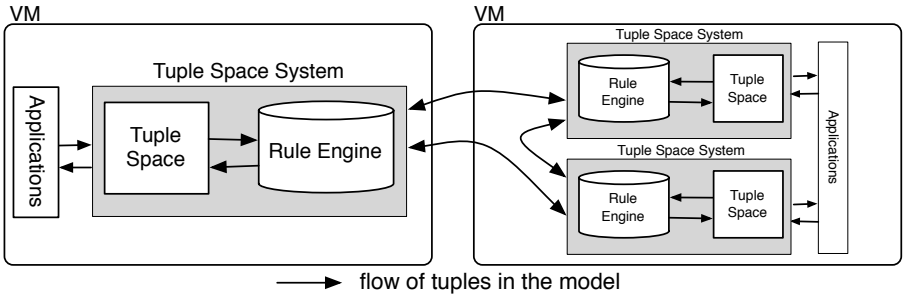

Figure 2: The TOTAM tuple space model.

The Core Model. The model underlying TOTAM tuples extends the notion of a traditional tuple space with machinery to control the scope and visibility of tuples. Figure 2 depicts our model. A device in the network corresponds to a virtual machine (VM) carrying one or more TOTAM tuple spaces. Each virtual machine forms a *TOTAM system*. TOTAM systems are interconnected by means of a MANET, forming a *TOTAM network*. The composition of a TOTAM network changes according to the network topology.

A TOTAM system consists of a tuple space, and a rule engine which infers when a tuple should be perceived by applications. The tuple space serves as the interface between applications and the TOTAM system. It only supports *non-blocking* Linda-like operations to insert, read and remove tuples. The main reason for the strict non-blocking operations is that it significantly reduces the impact of volatile connections on a distributed application. As an alternative to blocking operations, we provide the notion of a *reaction* to a tuple (similar to LIME reactions [1]): applications can register an observer that is asynchronously notified when a tuple matching a given template is read or removed from the tuple space.

The tuple space of a TOTAM system contains two types of tuples. *Public tuples* denote tuples that are shared with remote TOTAM systems, and *private tuples* denote tuples that remain local to the tuple space in which they were inserted and thus will not be transmitted to other TOTAM systems. Applications can insert private and public

tuples in the tuple space by means of the `out` and `inject` operation, respectively. As in LIME, applications can access tuples coming from the network without knowing the different collocated TOTAM systems explicitly.

Distribution of Tuples in the Network. When two TOTAM systems discover each other in the network, the public tuples contained in their tuple spaces are cloned and transmitted to the collocated TOTAM system according to a *propagation protocol* (reminiscent of TOTA propagation rules[5]). However, in contrast to TOTA, TOTAM triggers the propagation protocol *before* a tuple is being physically transmitted to a new TOTAM system, and *after* it is received by the new TOTAM system. Thus, the tuple itself contains all the information needed to dynamically adjust its scope in the TOTAM network while being propagated. In addition, such a scoping mechanism avoids unnecessary exchange of tuples and enhances privacy because the protocol can enforce that a tuple is not transmitted to a TOTAM system which is not in its scope. This differs from techniques in which the tuple space itself is scoped and tuples have to be inserted in a certain scope which can not be changed after the facts.

To be able to compute the scope of a tuple, each tuple space has a *tuple space descriptor*. This descriptor contains semantic information that is used by the tuples at sending time in order to decide whether a certain TOTAM system is in their scope. Descriptors are exchanged between two TOTAM systems when they meet for the first time or whenever a system decides to change its description.

Coordination. Our model combines replication of tuples for read operations (to support time-decoupled communication) while guaranteeing atomicity for removal operations (to support synchronization). In order for a remove operation to succeed, the system which created and injected the tuple in the network (called the *originator* system) needs to be connected. The underlying model does not remove tuples unless the originator system has acknowledged the operation. As such, a remove operation in our approach is executed atomically as defined in Linda [8]: if two processes perform a remove operation for a tuple, only one removes the tuple. When an originator is asked to remove one of its (stored) tuples by another system, it removes the tuple and injects an *antituple* for the removed tuple in the network. By means of antituples, TOTAM systems can "unsend" tuples injected to the network. For every tuple there is (conceptually) a unique antituple with the same format and content, but with a different sign. All tuples injected by an application have positive sign while their antituples have a negative sign. Whenever a tuple and its antituple are stored in the same tuple space, they *annihilate* one another, i.e., they both get removed from the tuple space.

Supporting Context-awareness. TOTAM tuples can carry a *context rule* that describes the runtime conditions under which the tuple is visible to an application, facilitating the development of context-aware applications deployed in a mobile environment. More precisely, a context rule is conceived as a set of conditions defined in terms of the presence of other tuples in the receiving tuple space. Such context rule is defined by the creator of the tuple and gets transmitted together with the tuple when the tuple is injected in the network. Defining context rules in terms of tuples allows the application to abstract away from the underlying hardware while keeping the simplicity of the tuple space model (both interactions and context information are defined using tuples).

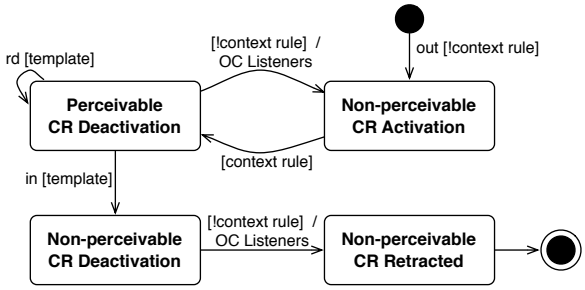

Figure 3: Lifespan of a context-aware tuple

When a tuple is inserted at a certain TOTAM system, the tuple is first handed over to the rule engine which installs the necessary machinery to evaluate the tuple's context rule. When the rule engine infers that the conditions on a context rule are satisfied, the tuple's context rule is triggered and the rule is said to be *satisfied*. Only when the context rule of a tuple is satisfied, is the tuple inserted in the tuple space of the TOTAM system. At that moment, the applications are able to read the tuple.

The rule engine plays a central role in our model as it takes care of reflecting the changes to the receiver's context so that applications cannot perceive those tuples whose context rule is not satisfied. In particular, it observes the insertion and removal of the tuples in the tuple space to infer which context rules are satisfied, and subsequently control which tuples present in the tuple space should be actually accessible by applications. As a result, programmers no longer need to infer the presence or absence of tuples manually as the rule engine takes care of it in an efficient way, making the code easier to understand and maintain.

The Lifespan of a Context-aware Tuple. Context rules introduce a new dimension to the lifespan of a tuple. Not only can a tuple be inserted or removed from the tuple space, it can now also be perceivable or non-perceivable for the application. Figure 3 shows a UML-state diagram for the lifespan of a *context-aware tuple*, i.e., a TOTAM tuple carrying a context rule. When an application inserts a tuple in a TOTAM system[1], the tuple is non-perceivable and its context rule is asserted by the rule engine. The rule engine then starts listening for the activation of that context rule (CR activation in the figure).

A tuple will become perceivable depending on whether or not the context rule is satisfied. If the context rule is satisfied, the tuple is perceivable and it is subject to tuple space operations (and thus becomes accessible to the application). If the tuple is non-perceivable, the tuple is not subject to tuple space operations but its context rule remains in the rule engine. Every time a tuple's context rule is not satisfied, the out-of-context listeners of a tuple (OC listeners in the figure) are triggered. Applications can install listeners to be notified when a tuple moves out of context.

[1]To keep the figure concise `out` denotes the insertion of a private or a public tuple.

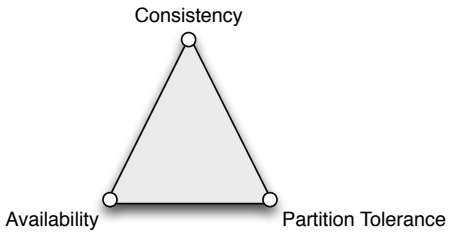

Figure 4: The CAP theorem.

Upon performing an `in` operation, the tuple is removed from the tuple space but its context is not modified. As such, the tuple is considered not to be perceivable (as it is no longer in the tuple space) and its context rule remains in the rule engine. Once out of the tuple space, the rule engine listens for the deactivation of the context rule. Once the context rule is no longer satisfied, the context rule is retracted from the rule engine, and the tuple will be eventually garbage collected.

A Best Effort Leasing Model. In order to support resource management in the face of permanent disconnections, we integrate a leasing model into public tuples. Designing a leasing model for a replication-based model, however, is challenging because tuples get replicated through a highly dynamic network topology. As such, at the moment a tuple needs to be removed it might not be possible to reach all systems where it was replicated to. The root problem is that a fundamental trade-off needs to be made between consistency and availability of data. Such a trade-off was first remarked in the domain of web services as Brewer's conjecture and later, formalized and proven as the CAP theorem in [11]. The theorem, depicted in Figure 4, states that it is impossible to simultaneously provide consistency, availability and partition tolerance; only two can be guaranteed at the same time. Since MANET applications must almost always be partition-tolerant because failures are inherent to the network topology, a choice needs to be made between providing consistency or availability. Relaxing consistency allows the system to remain highly available in the face of partial failures, while emphasizing consistency implies that the system may not be available under certain conditions. Since not having access to services can be considered the rule in a mobile setting, trading off consistency provides a scalable solution. Not only providing consistency usually introduces communication overhead, it may be also difficult (if not impossible) to achieve without making assumptions about the mobility of devices [9].

In TOTAM, we have introduced a leasing model which does not guarantee strong consistency, but only guarantees *best effort*. The underlying system tries hard to remove all replicas of a tuple, but does not give guarantees when this will happen. It is important to notice that the system does guarantee that the tuple is only removed once. All tuples are injected in the network with an associated lease denoting the total lifespan of a tuple. It is determined by the application that creates and injects the tuple to the network. Programmers can specify the lease time interval by means of dedicated support (explained later).

A lease is encoded as part of the propagation protocol, and as such, it is transmitted together with the tuple when it gets replicated and transmitted to another TOTAM system. When the lease term has elapsed, independently of the state in which a tuple is, the tuple becomes candidate for garbage collection in the TOTAM system. This means that the tuple's context rule is retracted from the rule engine, and the tuple is removed from the tuple space if necessary.

The transitions for leasing have been omitted from Figure 3 to keep it clear and concise. The lease of a tuple adds a transition from any state to the state representing that a tuple is non-perceivable and its context rule is retracted. When a public tuple gets removed as a result of an `in` operation, the underlying system sends an antituple to those systems that have previously received a replica of the removed tuple. If a TOTAM system cannot be reached, the removal of the tuple is delayed until its lease expires in the disconnected TOTAM system, or until it is in range with one of the TOTAM systems carrying its antituple.

4. Operational Semantics

Before describing TOTAM's programming API, we formalize our tuple space model by means of a calculus with operational semantics based on prior work in coordination [12, 13]. The syntax of our model is defined by the grammar shown in Table 1. k identifies the type of the tuple: $+$ for a public tuple, $\oplus$ for a private tuple and $-$ for an antituple. A tuple c is specified as a first order term τ. $\tau^k_{x,t}\langle r\rangle$ indicates that the tuple with content τ, type k and time interval of its lease t, originates from a tuple space with identifier x and is only perceivable when its context rule r is satisfied. The context rule is considered optional and the notation $\tau^k_{x,t}$ should be read as $\tau^k_{x,t}\langle 1\rangle$, i.e. the context rule is always true. The antituple of a tuple $\tau^k_{x,t}$ is denoted by $\tau^-_{x,t}\langle 0\rangle$, i.e. its context rule is always false.

$k ::= + \mid \oplus \mid -$	Tuple Types
$c ::= \tau^k_{x,t}\langle r\rangle$	Tuple
$S ::= \emptyset \mid c, S$	Tuple Set
$P ::= \emptyset \mid A.P$	Process
$C ::= \emptyset \mid (\llbracket S\rrbracket_x \mid C) \mid (P \mid C)$	Configuration
$A ::= out(x,\tau,r,t) \mid inject(x,\tau,r,t) \mid rd(x,\nu) \mid in(x,\nu) \mid$ $outC(x,\nu) \mid whenRead(x,\nu,P_a,P_d).P \mid whenIn(x,\nu,P_a,P_d).P$	Actions

Table 1: TOTAM Tuples: Grammar

A process P consists of a sequence of tuple space operations A. Tuples are stored in S which is defined as a set of tuples composed by the operator (**,**). A tuple space with content S and identifier x is denoted by $\llbracket S\rrbracket_x$. A system configuration C is modeled as a *set* of processes P and collocated tuple spaces $\llbracket S\rrbracket_x$ composed by the operator $|$. An application consists of all $P \in C$.

Next to the grammar, we assume the existence of a matching function $\mu(\nu,\tau)$ that takes a template ν and a tuple content τ, and returns θ. θ is a substitution map of

variable identifiers from the template ν to the actual values from τ. A concrete value in this map can be accessed by θ_z that returns the actual value for z. The matched tuple can be accessed by θ_τ. We also assume the existence of a function *time* which returns a numeric comparable value indicating the current time. $r(S)$ indicates that the context rule r is satisfied in the tuple set S.

The semantics of the our tuple space model is defined by the transition rules shown in Table 2. Every transition $C \xrightarrow{\lambda} C'$ indicates that a configuration C can be transformed into a configuration C' under the condition λ.

$$out(x,\tau,r,t).P|[\![S]\!]_x|C \xrightarrow{t'=time()+t} P|[\![\tau^{\oplus}_{x,t'}\ \langle r\rangle\ , S]\!]_x|C \quad \text{(OUT)}$$

$$inject(x,\tau,r,t).P|[\![S]\!]_x|C \xrightarrow{t'=time()+t} P|[\![\tau^{+}_{x,t'}\ \langle r\rangle\ , S]\!]_x|C \quad \text{(INJ)}$$

$$[\![\tau^k_{x,t}\ \langle r\rangle\ , S]\!]_x|[\![S']\!]_y|C \xrightarrow[(x\neq y)]{\tau\notin S'\wedge(k\neq\oplus)\wedge} [\![\tau^k_{x,t}\ \langle r\rangle\ , S]\!]_x|[\![\tau^k_{x,t}\ \langle r\rangle\ , S']\!]_y|C \quad \text{(RPL)}$$

$$rd(x,\nu).P|[\![\tau^k_{y,t}\ \langle r\rangle\ , S]\!]_x|C \xrightarrow[r(S)]{\mu(\nu,\tau)=\theta\wedge(k\neq-)\wedge} P\theta|[\![\tau^k_{y,t}\ \langle r\rangle\ , S]\!]_x|C \quad \text{(RD)}$$

$$[\![\tau^-_{y,t}\ \langle 0\rangle\ , \tau^k_{y,t}\ \langle r\rangle\ , S]\!]_x|C \xrightarrow{(k\neq-)} [\![\tau^-_{y,t}\ \langle 0\rangle\ , S]\!]_x|C \quad \text{(KILL)}$$

$$[\![\tau^k_{y,t}\ \langle r\rangle\ , S]\!]_x|C \xrightarrow{t\leq time()\wedge(k\neq-)} [\![\tau^-_{y,t}\ \langle 0\rangle\ , S]\!]_x|C \quad \text{(TIM)}$$

$$in(x,\nu).P|[\![\tau^k_{x,t}\ \langle r\rangle\ , S]\!]_x|C \xrightarrow[(k\neq-)]{\mu(\nu,\tau)=\theta\wedge r(S)\wedge} P\theta|[\![\tau^-_{x,t}\ \langle 0\rangle\ , S]\!]_x|C \quad \text{(INL)}$$

$$in(x,\nu).P|[\![\tau^+_{y,t}\ \langle r\rangle\ , S]\!]_x|[\![\tau^+_{y,t}\ \langle r\rangle\ , S']\!]_y|C \xrightarrow[(x\neq y)]{\mu(\nu,\tau)=\theta\wedge r(S)\wedge} P\theta|[\![S]\!]_x|[\![\tau^-_{y,t}\ \langle 0\rangle\ , S']\!]_y|C \quad \text{(INR)}$$

$$outC(x,\tau).P|[\![\tau^k_{y,t}\ \langle r\rangle\ , S]\!]_x|C \xrightarrow{!r(S)} P|[\![\tau^k_{y,t}\ \langle r\rangle\ , S]\!]_x|C \quad \text{(OC)}$$

$$whenRead(x,\nu,P_a,P_d).P|[\![S]\!]_x|C \xrightarrow{1} rd(x,\nu).P_a.outC(x,\theta_\tau).P_d|P|[\![S]\!]_x|C \quad \text{(WR)}$$

$$whenIn(x,\nu,P_a,P_d).P|[\![S]\!]_x|C \xrightarrow{1} in(x,\nu).P_a.outC(x,\theta_\tau).P_d|P|[\![S]\!]_x|C \quad \text{(WI)}$$

Table 2: Operational Semantics

The (OUT) rule states that when a process performs an `out` operation over a local tuple space x, the tuple is immediately inserted in x as a private tuple with context rule r and timeout t'. The process continuation P is executed immediately. When a tuple is inserted in the tuple space x with an `inject` operation as specified by (INJ), the tuple is inserted in x as a public tuple and is replicated to other tuple spaces as specified by (RPL). This rule states that when a tuple space y moves in communication range of a tuple space x, all tuples $\tau^k_{x,t}$ which are not private and are not already in y will be replicated to y. The (RD) rule states that to read a template ν from a tuple space x, x has to contain a matching $\tau^k_{y,t}$ and the context rule of τ is satisfied in S. (RD) blocks if one of these conditions is not satisfied. When (RD) does apply, the continuation P is invoked with substitution map θ. Note that we do not disallow x to be equal to y in this rule. The (KILL) rule specifies that when both a tuple τ and its unique antituple τ^- are stored in the same tuple space, τ is removed immediately. The (TIM) rule specifies that when the timeout of a tuple τ elapses, its antituple τ^- is inserted in the tuple space.

The `in` operation is *guaranteed* to be atomically executed. In the semantics, it has been split into a local rule (INL) and a remote rule (INR). (INL) works similarly to (RD), but it removes the tuple $\tau^k_{x,t}$ originated by the local tuple space x and inserts its antituple $\tau^-_{x,t}$. (INR) states that when the `in` operation is matched with a tuple published by another tuple space y, y must be one of the collocated tuple spaces (i.e. be in the configuration). Analogously to (INL), the tuple is removed and its antituple

is inserted. The (OC) rule states that to move out of context a tuple τ from a local tuple space x, x has to contain τ (possibly its antituple) and its context rule is *not* satisfied. The WR rule states that a `whenRead` operation performed on the local tuple space x with template ν and processes P_a and P_d, is immediately translated into a new parallel process and the continuation P will be executed. The newly spawned parallel process is specified in terms of performing a `rd` operation followed by an `outC` operation. A `rd` operation blocks until there is a tuple matching ν in the local tuple space. The continuation P_a is then executed to subsequently perform an `outC` which blocks until the tuple is no longer perceivable. Finally, the continuation P_d is invoked whereafter the process dies. The WI is specified analogously but as it models a `whenIn` operation, it performs a `in` operation rather than a `rd` one. The `wheneverRead` and `wheneverIn` operations have been omitted as they are trivial recursive extensions of `whenRead` and `whenIn`, respectively.

Note that $(KILL)$ does not remove antituples. This has been omitted to keep the semantics simple and concise. By means of (RPL), the antituple of a tuple τ is only replicated to those systems that received τ. In our concrete implementation if a system cannot be reached, the removal of the antituple is delayed until the timeout of its tuple elapses (which inserts an antituple as specified by (TIM)). An antituple can only be removed once there are no processes in the configuration which registered an `outC` operation on the original tuple.

5. Programming in TOTAM

In this section, we describe TOTAM from a programmer's perspective. We have implemented TOTAM as a middleware in AmbientTalk, a distributed object-oriented programming language specifically designed for mobile ad hoc networks [6]. Table A.4 in the Appendix summarizes TOTAM's programming API which we further detail in this section. We introduce the necessary syntax and features of the AmbientTalk language along with our explanation. We illustrate the set of operations provided by TOTAM by means of a concrete application called *Guanotes* that we use as running example throughout this section.

5.1. Running Example: the Guanotes Application

Guanotes is one of the default applications of *Urbiflock* [14]: a framework implemented in AmbientTalk for scripting social applications running on Android phones interacting via mobile ad hoc networks. It allow end-users to interact with their social networks (organized in *flocks*) by means of messages. A flock typically denotes a group of nearby users that match a number of user-defined criteria. Messages in Guanotes are called *guanotes*. A guanote can be sent to the users belonging to a flock which are currently in in their direct neighbourhood, or to all (reachable) users belonging to the target flock regardless of whether they are currently connected in the Urbiflock platform.

To exemplify the kinds of interactions using Guanotes, consider the following scenario: Alice and Bob are in the cafeteria of the university sports complex when they decide it would be nice to play some badminton. Since reserving the badminton field is

rather expensive, Bob decides to invite some extra players by taking his mobile phone and using Guanotes to send a message to the couples *currently* in the neighborhood who like to play badminton. Luckily, Carol and Denis who also wanted to play badminton see the invitation. They reply to Bob's message whereafter they meet and start playing a game. After the game, they get the wild idea to organize a badminton competition for next week. Again Bob takes his mobile phone and decides to send an invitation to *all* couples at the university who like badminton.

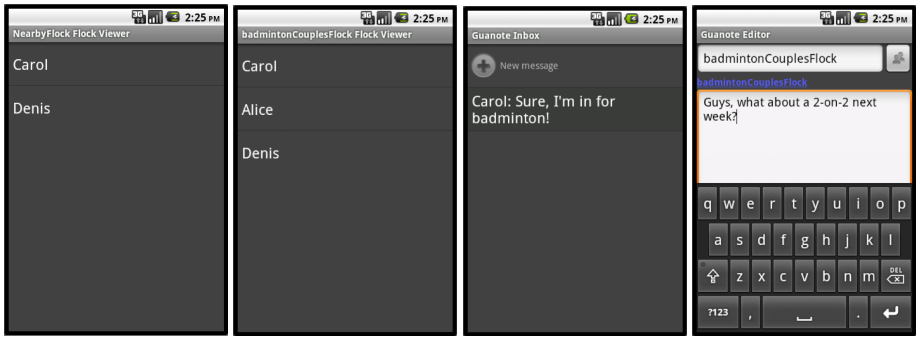

Figure 5: Bob's (a) NearbyFlock, (b) badmintonCouplesFlock, (c) Guanotes inbox and (d) Guanote editor

Figure 5 shows Urbiflock on Bob's device during the process of typing a guanote after playing a match with Carol and Denis. In particular, four different screenshots are displayed: (a) the contents of Bob's `NearbyFlock`, (b) the contents of Bob's `badmintonCouplesFlock`, (c) Bob's Guanotes inbox (that contains a guanote previously received from Carol replying to his first invitation), and (d) the editor for a new guanote to invite all his friend's couples to participate in a badminton tournament. Note that there are few users collocated at this time with Bob (people in the `NearbyFlock` shown in (a)) which belong to the `badmintonCouplesFlock` (b). As he would like to reach all couples interested in badminton for the tournament, he selects the `badmintonCouplesFlock` in the receiver list in Figure 5(d).

5.2. Defining and Inserting TOTAM Tuples

In order to create a TOTAM system and add it to the network, programmers can invoke the `makeTupleSpace` constructor function as follows:

```
def totam := makeTupleSpace(descriptor);
totam.goOnline();
```

This operation initializes a TOTAM system including the rule engine and the tuple space. Variables are defined with the keyword `def` and assigned to a value with the assignment symbol (:=). The newly created TOTAM system is then published in the TOTAM network by means of the `goOnline` operation. This operation returns a publication object with a `cancel` method to be able to remove the system from the TOTAM network. From then on, the newly created system perpetually looks for other systems in the TOTAM network and exchanges its descriptor with them. As explained before, the

descriptor contains semantic information relevant to the propagation of tuples. In the Guanotes application, the description is embodied in the user profile which is implemented as an isolate including fields representing the semantic information associated with the user (e.g., name, gender, hobbies, etc.).

We provide the `out(tuple)` operation to insert a private `tuple` in the tuple space. In order to insert a public tuple, thereby making it available to other collocated TOTAM systems, the **`inject:`** operation is provided. The code excerpt below illustrates the injection of the guanote depicted in Figure 5(d).

```
def aGuanote := tuple: [guanote, username, badmintonCouplesFlock,
  "Guys, what about a 2-on-2 next week?"];
totam.inject: aGuanote;
```

A tuple is created by means of the **`tuple:`** operation which takes as argument a list of fields (implemented in AmbientTalk as a table). As usual, the first field of a tuple is its type name. In this case, the tuple's type name is the `guanote` type tag[2]. A tuple representing a guanote consists of the sender of the tuple (stored in the `username` variable), the receiver target group (in this case the `badmintonCouplesFlock` group) and a textual message.

In its simplest form, the **`inject:`** operation takes as argument a tuple and injects the tuple into the network with a *default context rule* (i.e., the context rule is always true), and a *default lease* (i.e., a lease time interval predefined at the actor level). It returns a publication object with two methods: a `cancel` method that allows programmers to stop the propagation of the tuple in the TOTAM network, and a `retract` method that allows developers to "unexport" a tuple injected into the network (by making the system send an antituple to all TOTAM systems which received that tuple). A tuple carries a default propagation protocol that propagates it to all reachable TOTAM systems, exactly as in TOTA, i.e., it does not restrict the propagation of tuples. We explain how to encode and apply custom propagation protocols in Section 5.4.

Table A.4 shows the complete form of the **`inject:`** operation. The **`withLease:`** parameter takes an time interval denoting the specific duration of the lease associated with the tuple. The **`inContext:`** parameter takes a context rule defined as a table containing the set of templates and constraints that need to be satisfied for the tuple to be perceivable. Constraints are conceived as logical conditions on the variables used in a template. For example, consider that certain guanotes are only visible if the user is in, e.g., the cafeteria area. In order to model that a device is in a room, the application could inject a dedicated public tuple as follows:

```
totam.inject: (tuple: [inRoom, cafeteriaRoom])
inContext: [tuple: [location, ?loc], withinBoundary(roomArea,?loc)];
```

The `inRoom` tuple is a "helper" tuple which allows a guanote to determine when the user is in the cafeteria area. To this end, the `inRoom` tuple carries a context rule which consists of two terms that need to match:

[2]Type tags are a classification mechanism employed in AmbientTalk to categorize objects explicitly using a nominal type.

- First, there must be a tuple in the tuple space encoding the location of the user, i.e., matching the **`tuple:`** `[location,?loc]` template[3]. The `?` operator indicates a variable in a template. As shown in Table A.4, variables are actually defined using the `var:` construct which takes a symbol as argument. This paper will use the `?` operator in order to ease the reading. In our example, the template matches *any* `location` tuple in the tuple space.

- Second, the `location` tuple needs to satisfy a constraint: its coordinates have to be within the area of the room. The `withinBoundary` function returns such a constraint given the coordinates stored in the `?loc` variable and the cafeteria area stored in `roomArea` variable.

5.3. Reading and Removing Tuples from the TOTAM Network

In order to read and remove public tuples from the TOTAM network, programmers can use reactions to register a block of code that is executed when a tuple matching a template is inserted in the tuple space, reminiscent to LIME reactions. In what follows, we describe the 4 kinds of reactions supported (shown in Table A.4). The **`when: read:`** operation takes as argument a template to observe in the tuple space, and a closure that serves as a event handler to call when the tuple matching the template is available in the tuple space. It actually performs a reaction to a read operation, i.e., the matching tuple is not removed from the tuple space. In its simplest form, the **`when:read:`** operation only triggers the event handler once for a matching tuple. If several perceivable tuples match the template, one is chosen non-deterministically. The **`whenever:read:`** operation works analogously but it triggers the event handler for *every* perceivable tuple matching the template. The code excerpt below illustrates the usage of **`whenever:read:`** in the implementation of the Guanotes application.

```
def listenForGuanotesToOwner(guiListener) {
  def guanoteTemplate := tuple: [guanote, ?from, ?to, ?msg];
  totam.whenever: guanoteTemplate read: {
    guiListener<-guanoteReceived(from, to, msg);
  };
};
```

The `listenForGuanotesToOwner` function is called from the application GUI in order to start listening for tuples whose type name is `guanote`. When a perceivable tuple matches the template, the **`read:`** closure is applied binding all variables of the template to the values of the matching tuple. In this example, the application just extracts the information from the tuple and sends it to the GUI listener to be displayed[4].

Finally, the **`when:in:`** and **`whenever:in`** operations work analogously to the previous ones but, they perform a reaction to a removal operation rather than a read operation. In other words, these operations remove the tuple from the tuple space before applying the `closure` block. Recall that if the tuple to be removed comes from another TOTAM system, then the underlying TOTAM system has to contact the originator

[3] A template is created by means of the **`tuple:`** operation as well. However, only templates can take variables as fields.

[4] In the actual implementation, the tuple contents are converted into an AmbientTalk object which can be understood by the Java GUI using AmbientTalk's interoperability layer.

Listing 1: The TOTAM propagation protocol at sender side.

```
1 if: tuple.decideDie(tupleSpace) then: {
2   remove(tuple);
3 } else:{
4   if: tuple.inScope(senderDescriptor, receiverDescriptor) then: {
5     tuple := tuple.changeTupleContentOnSend();
6     transmit(tuple,locationOf(receiverDescriptor));
7   }
8 };
```

TOTAM system to atomically remove the original tuple. If that removal fails, the replicated tuple is not removed from the local tuple space and the `closure` is simply not triggered.

Our approach extends a LIME reaction with the notion of *context*: the event handler for a reaction can only be triggered when the tuple matching the pattern is perceivable. The complete forms of the previously described operations allows developers to react to a tuple moving out of context by installing an `outOfContext` listener. In the code snippet, the **`whenever:read:outOfContext:`** operation expresses that certain guanotes are only visible from within a certain room.

```
def inroomTemplate := tuple: [inRoom,?name];
totam.whenever: inroomTemplate read: {
  guiListener<-display("You are in room" + name);
} outOfContext: {
  guiListener<-display("You moved out of room" + name);
};
```

In this case, each time an `inRoom` tuple is matched, the application detects that the user moved in a certain room. Once the user leaves the room, the `inRoom`'s context rule is no longer satisfied and the `outOfContext:` closure is applied. In short, the extended versions of `when(ever):*` operations asynchronously apply the first closure when the tuple is perceivable, and the **`outOfContext:`** closure when the context rule of the matching tuple is not satisfied.

Apart from the reactions previously explained, we also provide the `rdp` and `rdg` operations to read one and all tuples in the tuple space, respectively, that match a given template (at the point in time when the operation is executed).

5.4. Writing Application-Specific Propagation Protocols

As previously mentioned, tuples hop from one TOTAM system to another according to a propagation protocol carried by the tuple itself. A propagation protocol is encoded in TOTAM as a *pass-by-copy* object implementing a number of methods which are called by the the underlying TOTAM system *before* and *after* transmitting a tuple. In this section, we explain the API of TOTAM's propagation protocols and how programmers can define custom propagation protocols.

Listing 1 shows the protocol before propagating the tuple (i.e., at sender side). First, `decideDie` is called. It returns a boolean indicating whether or not the tuple should be removed. If the protocol decides not to remove the tuple, the protocol will be asked

whether a potential receiver is in the scope of this tuple. `inScope` takes as argument the descriptor of the sender and receiver tuple space. The protocol can then decide to propagate the tuple to the receiver descriptor. Finally, before transmitting the tuple, the protocol can change its content (line 5).

Listing 2 shows the protocol on the receiver side. A tuple is first asked whether it should enter the receiving tuple space (line 1). It will then be allowed to execute some action on the local tuple space (line 2). Finally the tuple can change its content and decide to notify the local tuple space that it arrived (line 3 and 4). Tuples with a protocol which do not notify the local tuple space can be used e.g., to remove inappropriate tuples.

The code excerpt below illustrates how the Guanotes application creates a custom propagation protocol to be carried by guanotes which should only be received by users within direct communication range, i.e., systems only one hop away.

```
1  def makeGuanoteToNearbyUsers(from, msg){
2    def oneHopProtocol := propagationProtocol: {
3      def hops := 0;
4      def inScope(senderTs, senderDes, receiverDes) {
5        hops < 1;
6      };
7      def changeTupleContentOnReceive(ts) {
8        hops := hops + 1;
9        self.getContent();
10     };
11   };
12   tuple: [guanote, from, to, msg] withPropagationProtocol: guanotesProtocol;
13 };
```

`propagationProtocol:` is an operation that allows the programmer to quickly create a custom protocol. It expects as argument a code block that is used to extend the default propagation protocol prototype (which corresponds to a tuple which always propagates to every tuple space encountered). In this example, the `inScope` and `changeTupleContent` methods are overriden in order to limit the propagation of the guanote. The `inScope` method will verify that the tuple has been transmitted only one hop. The `changeTupleContent` increments the hop counter. Line 12 shows programmers can associate a propagation protocol with a tuple by means of the **`tuple:withPropagationProtocol:`** construct.

Listing 2: The TOTAM propagation protocol at receiver side.

```
1  if: tuple.decideEnter(ts) then: {
2    tuple.doAction(ts);
3    tuple := tuple.changeTupleContentOnReceive();
4    if: tuple.decideStore(ts) then: {
5      ts.out(tuple);
6    };
7    tuple
8  } else: { nil };
```

6. Developing Mobile Applications with TOTAM

We have build several applications in TOTAM to showcase the suitability and applicability of its programming API. TOTAM has also been used as a teaching platform in the Master in Computer Science curriculum at the Vrije Universiteit Brussel, for a course on mobile and distributed computing. Students have used TOTAM for simple exercises as well as programming projects implementing e.g., multi-player mobile games such as a urban game for collaborative noise mapping[5] Not only has application development served us to validate programming abstractions proposed in TOTAM, but it has also uncovered the real needs of developers.

In this section, we elaborate on two applications that benefit from using TOTAM to enable communication in a mobile setting in which devices interact via wireless ad hoc connections. The first one involves the ability to detect and react to changes in the application context, while the second one involves the ability to perform collaborative tasks in the presence of disconnections. We conclude the section by describing the use of TOTAM in an industrial case with the Brussels public transport company.

6.1. Flikken

Flikken[6] is a game in which players equipped with mobile devices interact in a physical environment augmented with virtual objects. Flikken is a significant subset of an augmented reality game inspired by the industrial game "The Target"[7]. The game consists of a dangerous gangster on the loose with the goal of earning 1 million euro by committing crimes. In order to commit crimes the gangster needs to collect burglary tools around the city (knives, detonators, etc). Policemen work together to shoot the gangster before he achieves his goal.

Figure 6 shows the gangster's as well as a policeman's mobile device at the time the gangster has burgled the local casino. The gangster knows the locations with larger amounts of money (banks, casinos, etc). When a gangster commits a crime, policemen are informed of the location and the amount of money stolen. Policemen can see the position of all nearby policemen and send messages to each other in order to coordinate their movements. The gangster and policemen are frequently informed of each other's positions and can shoot at each other.

Flikken epitomizes MANET applications that react to context changes in the environment. Examples of such changes include player's location, appearance and disappearance of players, and the discovery of virtual objects while moving about. Moreover, how to react to these changes highly depends on the receivers of the contextual information. For example, virtual objects representing burglary tools should only be perceived by the gangster when he is close their location while they should not be perceived by policemen at all. In what follows, we describe the coordination and interaction between policemen and the gangster using tuples.

---

[5] Demos of the best student projects of the latest academic years are available at `http://www.youtube.com/playlist?list=PL71615F77073CD26C&feature=plcp`

[6] Flikken means *cops* in Flemish.

[7] `http://www.lamosca.be/en/the-target`

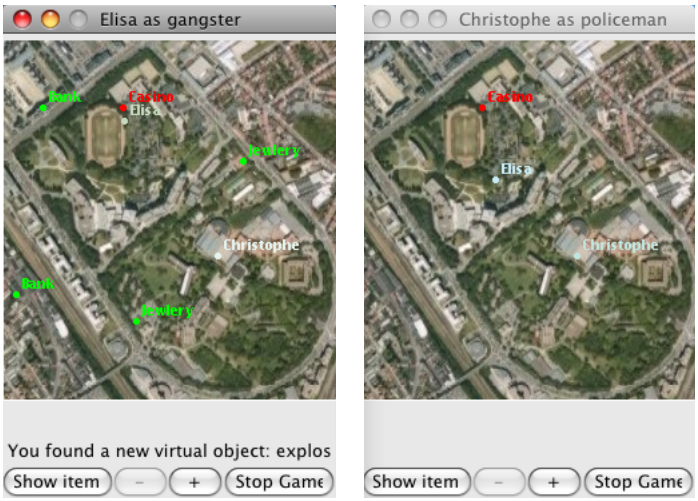

Figure 6: Flikken GUI on the gangster device (left) and a policeman device (right).

6.1.1. Design and Implementation

Every player has a TOTAM system. Once the game starts, policemen and gangster communicate by means of the TOTAM network. Throughout the city various context providers (i.e., TOTAM systems) are placed playing the role of virtual objects or crime locations by broadcasting the necessary tuples. A special type of context provider is at the headquarters (HQ) of the players which signals the start of the chase.

In this section, we only describe the set of tuples coordinating the core functionality which counts 11 tuples (6 of which carry a custom context rule, and 2 a custom lease) and 7 reactions. Table 3 shows an overview of the tuples used in the game. The tuples are divided into five categories depending on the entity that injects them in the environment, i.e., all players, only gangster, only policemen, headquarters and city context providers. A tuple is denoted by the term τ and the first element of a tuple indicates its type name. We use capitals for constant values.

The TOTAM system on each player's device is identified by a descriptor including its username and the team that he belongs to. However, since there is communication between policemen and the gangster, tuples injected by players are not scoped on a team basis. We make use of context rules to control the perception of certain tuples when necessary (e.g., to make sure the gangster does not see certain tuples). In contrast, all tuples injected by the TOTAM system at the headquarters are scoped on a team basis using the player tuple space descriptor.

The player's TOTAM system carries a private tuple τ(TeamInfo, uid, gip) indicating which team he belongs to. Every player injects his location to the TOTAM network by means of the tuple τ(PlayerInfo, uid, gip, location). This tuple is injected into the tuple space with a custom lease as we will explain later. Both the `PlayerInfo` and `TeamInfo` tuples are often used in other tuple's context rules to identify the current whereabouts of a player and his team. For example, the tuple representing a grenade

Tuple Content	Tuple Context Rule / Lease	Tuple Description
	All Players	
τ(TeamInfo, uid, gip)	[true]	Private tuple denoting the player's team.
τ(PlayerInfo, uid, gip, location)	[τ(TeamInfo, ?u, ?team), ?team $\neq$ gip] / [6 min]	Injected to opposite team members every 6 minutes to notify the position of a player. Location is a 2-tuple indicating the (GPS) coordinates of the player.
τ(OwnsVirtualObject, GUN, bullets)	[true]	Private tuple inserted by players when they pick up their gun at their HQ.
	Only The Gangster	
τ(CrimeCommitted, name,location,reward)	[τ(TeamInfo, ?u, POLICEMAN)]	Notifies policemen that the gangster committed a crime.
τ(OwnsVirtualObject, type, properties)	[true]	Private tuple inserted when the gangster picks up a virtual object in the game area.
	Only Policemen	
τ(PlayerInfo, uid, gip, location)	[τ(TeamInfo, ?u, gip)] / [1 min]	Notifies the position of a policemen to his colleagues every time he moves.
	Headquarters	
τ(InHeadquarters, location)	[τ(PlayerInfo,?u,?team,?loc), inRange(location, ?loc)]	Notifies that the player entered his HQ. Used to start the chase (when this tuple moves out of context for the gangster's HQ) and to reload policemen's guns.
τ(CrimeTarget, name, location)	[true]	Notifies the gangster of the position of crime targets.
τ(CommitCrime,name, location,reward,vobj)	[τ(PlayerInfo,?u, GANGSTER,?loc), inRange(location,?loc), hasVirtualObjects(vobj)]	Notifies the gangster of the possibility of committing a crime. `hasVirtualObjects` takes an array of virtual object ids and checks that the gangster has the required `OwnsVirtualObject` tuples.
	City Context Providers	
τ(VirtualObject, id, location)	[τ(TeamInfo, ?u, GANGSTER), τ(PlayerInfo,?u, GANGSTER,?loc), inRange(location, ?loc)]	Notifies the gangster of the nearby presence of a virtual object. `inRange` is a helper function to check that two locations are in euclidian distance.
τ(Rechargeable-VirtualObject,GUN, BULLETS)	[τ(InHeadQuarters,?loc), τ(OwnsWeaponVO,GUN,?bullets), ?bullets $<$ BULLETS]	Represents the player's gun. The gangster gets only one charge at the start of the game, while policemen's guns are recharged each time they go back to their HQ.

Table 3: Overview of the Tuples used in Flikken

uses these tuples as follows.

```
totam.inject: (tuple: [VirtualObject, grenade, location]);
inContext: [tuple: [TeamInfo, ?u, GANGSTER],
            tuple: [PlayerInfo, ?u, GANGSTER], ?loc],
            inRange(location, ?loc) ]
```

The tuple τ(VirtualObject, grenade, location) should be only visable if the receiver is a gangster whose location is physically proximate to the virtual object. The `inRange` function checks whether the gangster location (given by `?loc` in the `PlayerInfo` tuple) is in euclidian distance with the location of the grenade (stored in `location`). Upon removal of a `VirtualObject` tuple, a private tuple τ(OwnsVirtualObject, object) is inserted in his tuple space. `CommitCrime` tuples notify the gangster of a crime that can be committed. As crimes can only be committed when the gangster has certain burgling items, the context rule of the `CommitCrime` tuple requires that certain `OwnsVirtualObject` tuples are present in the tuple space. For example, in order for the gangster to perceive the `CommitCrime` tuple for the `grandCasino`, a τ(OwnsVirtualObject,grenade) tuple is needed as shown below.

```
totam.inject: tuple(CommitCrime, grandCasino, location, reward)
```

```
inContext: [tuple: [TeamInfo, ?u, GANGSTER],
            tuple: [PlayerInfo, ?u, GANGSTER, ?loc],
            inRange(location, ?loc),
            tuple(OwnsVirtualObject,grenade)];
```

Note that since context rules can be developed separately, this enables programers to reuse rules to build different kinds of tuples, increasing reusability. As shown above, we have reused the `inRange` function to build both the rule for the different `VirtualObject` tuples and `CommitCrime` tuples. Decomposing a tuple into content and context rule also leads to separation of concerns, increasing modularity.

Each player also registers several reactions to (1) update his GUI (e.g., to show the `OwnsVirtualObject` tuples collected), and (2) inject new tuples in response to the perceived ones. For example, when a gangster commits a crime, he injects a tuple τ(CrimeCommitted, name,location,reward) to notify policemen. The code below shows the reaction on `PlayerInfo` tuples installed by the application.

```
def playerinfoTemplate := tuple: [PlayerInfo, ?uid, ?tid, ?location];
totam.whenever: playerinfoTemplate read: {
  GUI.displayPlayerPosition(tid, uid, location);
} outOfContext: {
  def matchingPlayerinfoTemplate := tuple: [PlayerInfo, uid, tid, ?loc];
  def tuple := totam.rdp(matchingPlayerinfoTemplate);
  if: (nil == tuple) then: { GUI.showOffline(uid) };
};
```

Whenever a `PlayerInfo` tuple is read, the player updates his GUI with the new location of that player. As `PlayerInfo` tuples are injected with a custom lease, they are automatically removed from the tuple space after their time interval elapses triggering the **`outOfContext:`** handler. In particular, opposing team members receive the player's location with a lease of 6 minutes, and policemen share their location with a lease of 1 minute. In the example, the **`outOfContext:`** handler grays out the GUI representation of a player if no other `PlayerInfo` tuple for that player is in the TOTAM system. If the `rdp` operation does not return a tuple, the player is considered to be offline as he did not transmit his coordinates for a while.

Note how the integration of context into reactions avoids having to write imperative code for inferring tuple perception. The underlying rule engine takes care of it, making the code easier to understand and maintain.

6.2. WeScribble

WeScribble is a collaborative drawing application that allows users to dynamically participate in a drawing session with other people nearby[8]. The application assumes no other infrastructure than mobile devices and wireless ad hoc connections between these devices. Users can join a drawing sessions by contacting any user which is already part of a drawing session. When a user joins a drawing session, he can add, remove or move shapes on the canvas of that session which is *virtually* shared between the different participants. Figure 7 shows a screenshot of a weScribble drawing session running on Android mobile devices. We will call a participant in a drawing session a *drawer*. In its

[8] A demo of the weScribble application is available at `http://www.youtube.com/watch?v=k0HYqRCxtHc`

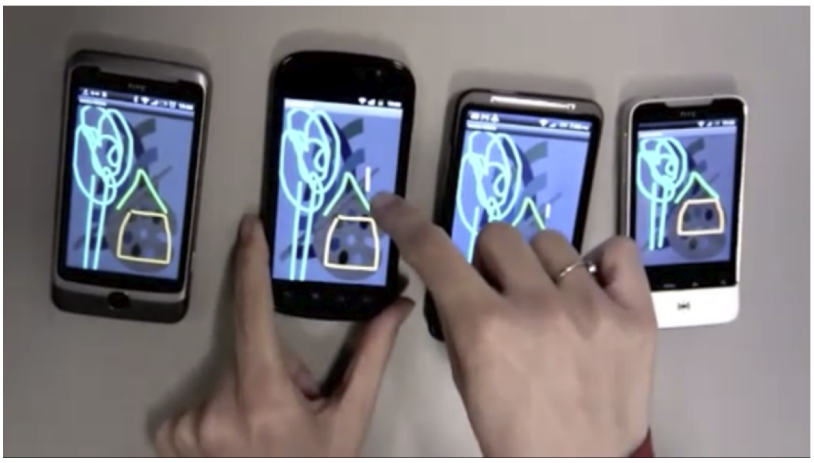

Figure 7: weScribble drawing session over four Android devices.

extended version, users can draw several kind of shapes (e.g. rectangle, square, line), move them in the canvas, and change their color. Users can also select one or more shapes and group them in a *grouped shape*.

6.2.1. Design and Implementation.

The application (supporting rectangle, circles and lines as shapes) consists of 12 types of tuples, 10 **`read`** reactions and 2 **`in:`** reactions. The basic idea is that the canvas is represented by a tuple space where devices can read and post `Shape` tuples representing shapes being drawn. The main challenges in the design of the application is to detect nearby drawers, the disconnection of drawers, and updates on group shapes.

Since TOTAM follows a replicated-based tuple space approach, it is important to avoid that drawers receive outdated tuples from other drawers whose devices are not nearby, i.e., devices currently not connected to the TOTAM network. To this end, the application tuple space descriptor makes use of a drawing session id. When a user creates a drawing session, it places a tuple representing the session id, which can be read by all users. Users joining a session first need to read a `DrawingSession` tuple and then update their tuple space descriptor so that they only share `Shape` tuples for that session. This application showed the importance of providing support for changing the tuple space descriptor dynamically which is for example not necessary for the Flikken application. We will further discuss the implication of changing tuple space descriptors in Section 8.

To ensure that players have access to the shapes that he would like to modify created by other players, the application employs an `in` reactions on the shape tuple it needs to modify. If the tuple is not returned, it means that the drawer who first created the shape has disconnected so the selection of the shape does not succeed. If the tuple is returned, it means that the drawer is allowed to modify the shape and it becomes temporary owner of the shape. In order to detect disconnection of drawers, each drawer injected

a `Drawer` tuple in the network with a lease of 30 seconds. If the `Drawer` tuple out of context listener was triggered and a more recent `Drawer` tuple is not present in the tuple space, the drawer is considered to be disconnected, and its shapes gets grayed out in the GUI.

In order for a drawer to create a group shape, the different shapes forming the group should be accessible so that they can be bundled together into one group shape. For this reason, the implementation injects a `GroupShape` tuple whose context rule specifies the presence in the tuple space of the individual shapes, and register a read reaction on the `GroupShape` tuple so that the shapes can be merged and the GUI is notified. The use of such a context rule is critical to deal with consistency issues. Note that when a new drawer joins the drawing session, it receives the individual `Shape` tuples and the `GroupShape` tuple in its tuple space. However, the order in which those tuples are read is nondeterministic, so if the `GroupShape` tuple is read before any of the individual `Shape` tuples, the application will try to merge unavailable `Shape` tuples. By employing a context rule on the `GroupShape` tuple, developers are sure that only when all the shapes are available the group will be formed.

6.3. Industrial Case Study

TOTAM has been used to prototype a urban bulletin application in collaboration with the STIB, Brussels' public transport company. The application works like the bulletin boards that one sometimes sees at the exit of supermarkets with messages such as "Im looking for a housekeeper", but integrated in a small part of the Brussels public transportation system. Passengers can use an Android device to post messages on the bus and read messages that were posted by previous passengers. Crossing buses exchange messages such that they get percolated through the transportation network.

Our experiments were conducted in May 2011 in a MIVB depot and included 3 buses which were equipped with customized onboard computers. Figure 8 shows a photo of the setup. Each bus carries a onboard computer (Pentium M-compatible with 1GB RAM) with a 802.11a interface attached to the computer which provides DHCP connections. Each bus also has a GPRS modem attached to the computer. The onboard computer runs Linux Debian 5.0 and a dedicated AmbientTalk distribution. This modified AmbientTalk ables simultaneous connections to two different network interfaces. This was required in order to enable both customer-to-bus transfers and bus-to-bus transfers. Passengers were equipped with stock Android devices; we employed Google Nexus S, HTC Desire, and HTC Sensation phones running Android 4.0.

Each passenger and bus carry a TOTAM tuple space representing the bulletin board. Using an unoptimized TOTAM and AmbientTalk interpreter, customer-to-bus tuple transfer took up till 3 seconds. Connections between the onboard wifi antenna and smartphone are stable until the distance between the smartphone and the bus is increased to 60 m. Such a distance causes significant packet loss or even disconnections, strongly degrading the QoS. A demo of the customer-to-vehicle mobile bulletin board application is available at `http://www.youtube.com/watch?v=N7mxaPftod4`.

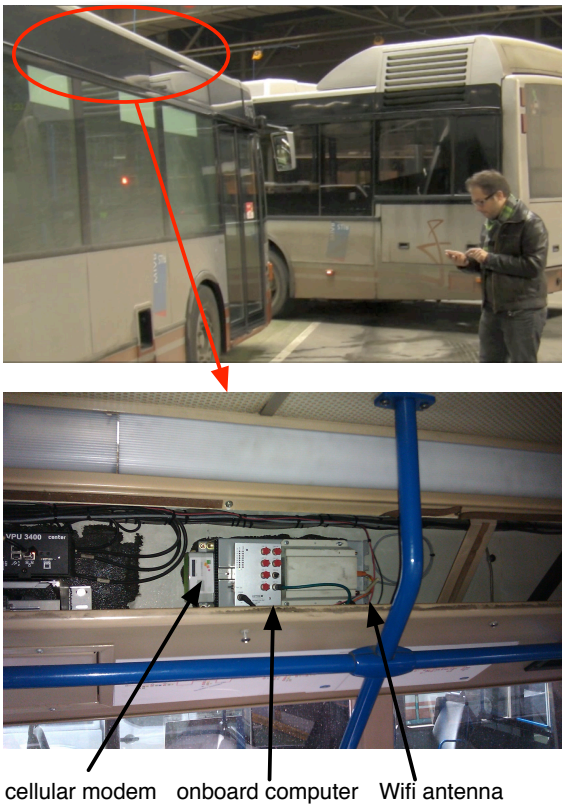

Figure 8: Photos of a deployed bus.

7. Software Engineering Analysis

The goal of this section is to highlight how TOTAM features aid the development of context-aware applications running on mobile ad hoc networks from a software engineering point of view.

7.1. Context Representation

One of the main benefits of TOTAM is that it provides a form of context representation in which a tuple itself can determine the runtime conditions in which a receiving application should be in order to perceive the tuple. By introducing a rule engine into the tuple space system, the underlying system takes care of making accessible context information only when it is valid for the application's context situation. A tuple space model with such abstractions has the following benefits:

1. Decomposing a tuple into content and context rule leads to separation of concerns, increasing modularity.

2. Since context rules can be developed separately, it enables programers to reuse the rules to build different kinds of tuples, increasing reusability. For example, in Flikken, we used a `inRangeOfGangster(?loc)` function to build the rule for the different `VirtualObject` tuples which was also reused to build the three first conditions of `CommitCrime` tuples.
3. Programmers do not need to add computational code to infer tuple perception as the rule engine takes care of it in an efficient way, making the code easier to understand and maintain.

7.2. Dealing with Partial Failures

Another benefit of TOTAM is providing a tuple space model that deals with the effects of partial failures inherent to a mobile ad hoc network setting. By default, applications are not aware of the intermittent disconnections of other TOTAM systems in the network since the model abstracts the configuration of the network. When a higher degree of context-awareness is required, tuples can contain context rules describing the runtime conditions under which the tuples should be visible in the receiving tuple space. In order to deal with permanent disconnections, programmers can inject tuples with a *lease* which determines how long the tuple should remain in the tuple space.

Encoding leases as part of a tuple's propagation protocol relieves programmers of manually encoding when a tuple should expire in the propagation protocol itself. Doing so is challenging because tuples are replicated amongst the network and developers need to take into account clock synchronization. Keeping clocks synchronised is a well known problem in distributed systems [15], but this issue is exacerbated in a highly disconnected environment since the system cannot provide strong guarantees about the expiration time. In other words, it could be that some copy of a tuple is still unrightfully active in the system, causing the tuple to be perceivable by applications. The developer must then add boilerplate code to ascertain whether the tuple is expired or not. Because we encoded leases into the tuple space model, the system takes care of the issues of clock synchronization. At worst, the asynchrony causes a tuple to be subject to a **`read`** operation. However, the tuple will not be available for `in` operations because for a `in` operation to succeed the owner of the tuple has to be contacted. When contacting the owner, the requesting tuple space will be informed that the lease's tuple is expired.

8. Evaluation

To complete the above software engineering analysis, it is also important to analyze the effectiveness of TOTAM from a distributed system point of view. By making use of tuple space descriptors, programmers can scope their tuples preventing them to be transported to unwanted locations. These descriptors are crucial to provide programmers with a hook to encode privacy strategies. For example, although the descriptors in the Flikken are limited to be simple objects carrying the team identifier, this can be extended to compute an encryption challenge. By avoiding the unnecessary tuple transportation, our approach can minimize network traffic.

Tuple space descriptors are exchanged between two locations when they meet for the first time and whenever a location decides to change its description. In case descriptors stay constant and prevent the propagation of tuples they can drastically reduce the

burden on the network. In the other case when descriptors change a lot or do not prevent the transportation of tuples the danger exists that the network traffic gets dominated by the transmission of descriptors. In the remainder section, we evaluate when the use of tuple space descriptors is beneficial and in which cases it is not in terms of network traffic.

8.1. Worst Case

In the worst case there is one message that has to be transported to all connected locations. This means that the exchange of the tuple space descriptors is an overhead as the tuple was unlimited in its scope i.e. the tuple floods the network. The network traffic generated for this tuple to be sent over the network when two locations meet can be computed as follows. Every location x connecting to a location y will first receive the descriptor D_y over the network and then receive the tuple t_{x1}. The total amount of network traffic for this tuple can thus be computed by summing the exchange of all descriptors with the total amount of exchanged tuples. This is shown in the following equation where n represents the number of connected locations.

$$NetworkTraffic = (\sum_{x=0}^{n}\sum_{y=0}^{n} D_y) + n.t_{x1} \tag{1}$$

In case the descriptors do not change they will only be transported once when two locations discover each other. This means that from the second communicated tuple the cost of the descriptor is dropped in the above equation. The resulting equation is exactly the traffic that is normally transferred ($n.T_{x1}$) when not making use of tuple space descriptors. However in the worst case all connected locations change their descriptors for every transmitted tuple. In this case the overhead of transferring the descriptors is given by the following equation where N is the number of transferred tuples.

$$NetworkOverhead = (\sum_{x=0}^{n}\sum_{y=0}^{n} D_y) * N \tag{2}$$

The overhead of exchanging the descriptors will be quadratic to the number of connected locations over time. This clearly shows that when descriptors change a lot and tuples have to be sent to all connected locations encountered it is not beneficial to use tuple space descriptors.

8.2. Best Case

In the best meaningful case[9] the sent tuples are sculpted to be only sent to one location and the tuple space descriptors do not change over time. We illustrate this case by a tuple that hops from location to location in a ring. In every hop the tuple adjusts its scope to the next hop in the ring configuration. We have illustrated the network traffic generated by this scenario for TOTAM and traditional approaches in

[9]The theoretical best case is when the tuple is meant for nobody and thus does not generate network traffic at all.

Figure 9. It is important to split up the network traffic generated by TOTAM in the case for the first tuple and the successive ones. As can be observed on the top left of the figure, before sending the first tuple over the ring all descriptors have to be exchanged. However, when this tuple is further propagated over the ring exchanging the descriptors is not necessary anymore so the number of exchanged tuples for one round equals the size of the ring (as shown in the second and third step of the figure). This is in contrast to approaches where the tuples are not scoped, in these cases the number of exchanged tuples for one round is quadratic to the number of locations participating in the ring.

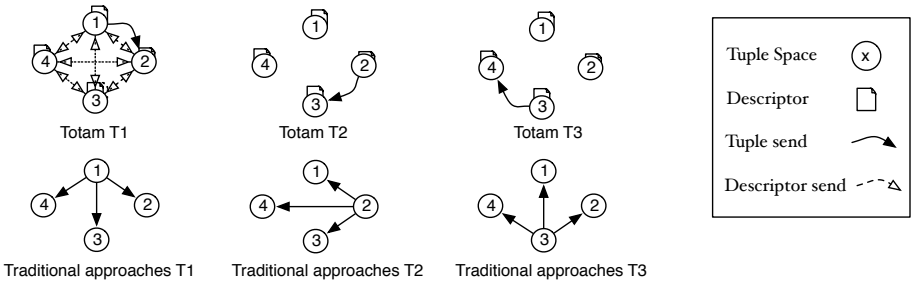

Figure 9: Set-up of best case scenario

8.3. Benchmarks

We now report on micro-benchmarks of our TOTAM prototype implementation. In order to validate whether the use of tuple space descriptors helps the programmer to decrease the network traffic in his applications we have benchmarked two implementation of the ring example shown in the previous section. A first implementation makes use of a tuple space descriptor that corresponds with the number of the node in the ring where the tuple is jumping to. The relevant code of the protocol that implements this protocol is shown in figure 3. The second implementation does not make use of tuple space descriptors and thus corresponds to traditional tuple space based approaches. Therefore, the tuple is sent to each of the participants where it is stored in the memory of the receiving participant. As argued in the previous section such an implementation generates a considerable amount of network traffic.

Listing 3: TOTAM ring descriptor.

```
def protocol(ringSize) {
  propagationProtocol: {
    def inScope(senderDes, receiverDes) {
      receiverDes ==  (senderDes+1)%ringSize
    };
  };
};
```

In our tests, first a ring of ten nodes is created, then a single tuple is inserted that jumps from node n in the ring to node $n + 1$. The time in function of the number of hops the tuple jumps is shown in figure 10. The results depicted were obtained on an Intel Core 2 Duo with a processor speed of 2.53 GHz running Mac OS X (10.8.3). As

shown in this figure, the implementation with tuple space descriptors is significantly faster than the implementation without tuple space descriptors. The figure shows that the overhead of not making use of tuple space descriptors becoming visible after only a couple of hops. The main reason why the overhead becomes noticeable very fast, is that not only the network traffic is significantly higher, the processing time for matching tuples is also significantly higher. All the received tuples of the nodes in the ring are processed one by one and while propagating nodes in the network all the tuples that do not match still need to matched against the event handlers.

In conclusion, our benchmarks confirm that the TOTAM tuple space implementation can significantly reduce the network traffic. This results in a more responsive system, for twenty hops a 5 time speedup was measured. Moreover, as there are less tuples received the individual nodes in the network have less processing work to perform.

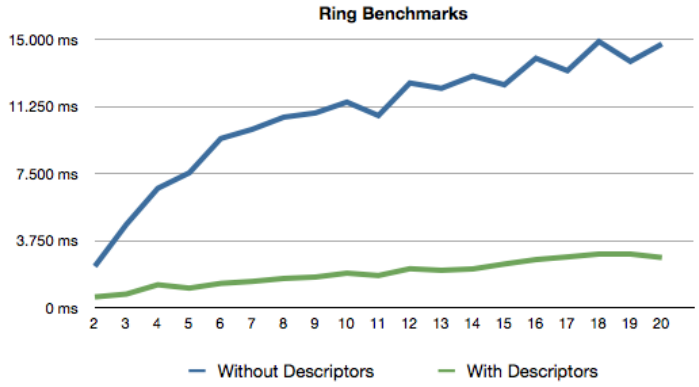

Figure 10: Benchmark comparison of tuple propagation in a ring structure.

8.4. Summary

The use of tuple space descriptors in combination with scoped tuples has the potential to drastically reduce the network traffic when 1) the tuples will be prevented from hopping to other locations and 2) the descriptors do not change often relative to the number of tuples in the system. However, how often do the descriptors change is highly application-dependent. For example, in Flikken, tuple descriptors do not change over time because players were not allowed to switch teams. In Guanotes, the tuple descriptor changes when the end-user updates its user profile. As future work, we would like to conduct experiments with devices to measure the costs of changing tuple space descriptors for different applications as well as for different propagation strategies.

9. Prior Work

As mentioned in the introduction, the TOTAM system described in this paper is an updated version of the tuple space system with the same name presented in previous

work [7, 16]. In this section, we highlight how the designed of the updated TOTAM defers from its predecessor.

In the first version of TOTAM, not all tuple space operations were designed to be non-blocking. However, from other previous work [17], we have found that a loosely-coupled communication model mitigates the effects of the hardware characteristics inherent to mobile ad hoc networks. As such, TOTAM's updated version was re-designed from the ground up not to support blocking operations.

In addition, tuples were merely wrappers on objects rather than first-class values. As a result, the design of tuples did not preserve encapsulation and the code for the propagation protocol was interwoven with the content of the tuple, hindering the reuse of protocols for other tuples and applications.

Finally, the presented TOTAM proposes a redesigned propagation protocol API solving two main limitations that the initial API suffered from. First, the `inScope` operation did not include the descriptor of both the sender and receiver tuple space. This forced developers to manually update changes on the sender descriptor when a tuple arrived to a tuple space by misusing `doAction` operation. Second, the first propagation API did not allow to change the tuple content before being sent and received in a tuple space.

10. Related Work

In this section, we discuss related work with regard to the various concepts integrated in TOTAM, namely its support for context-awareness, its scoping mechanism and leasing model. Finally, we compare our approach to publish/subscribe middleware which is closely related to tuple space model as it provides similar decoupling properties [18].

Context-Awareness. We now discuss related tuple space systems modeled for context-awareness and show how context-aware tuples differ from them. In TOTA, tuples themselves decide how to replicate from node to node in the network. Because tuples can execute code when they arrive at a node, they can be exploited to achieve context-awareness in an adaptive way. However, programming such tuples has proven to be difficult [5]. TOTA, therefore, provides several basic tuple propagation strategies. None of these propagation strategies addresses the tuple perception problems tackled by our approach. Writing context-aware tuples in TOTA would require a considerable programming effort to react on the presence of an arbitrary combination of tuples as it only allows reactions on a single tuple.

GeoLinda [19] augments federated tuple spaces with a geometrical read operation `read(s,p)`. Every tuple has an associated shape and the `rd` operation only matches those tuples whose shape intersects the addressing shape `s`. GeoLinda has been designed to overcome the shortcomings of federated tuple spaces for a small subset of potential context information, namely the physical location of devices. As such, it does not offer a general solution for context-aware applications. In contrast, we propose a general solution based on context rules, which allows programmers to write application-specific rules for their tuples. Moreover, in GeoLinda the collocation of

devices still plays a central role for tuple perception which can lead to erroneous context perception.

EgoSpaces provides the concept of a *view*, a declarative specification of a subset of the ambient tuples which should be perceived. Such views are defined by the *receiver* of tuples while in context-aware tuples it is the other way around. Context-aware tuples allow the *sender* of a tuple to attach a context rule dictating the system in which state the receiver should be in order to perceive the tuple. EgoSpaces suffers from the same limitations as federated tuple spaces since, at any given time, the available data depends on connectivity [4].

The Fact Space Model [20] is a LIME-like federated tuple space model that provides applications with a distributed knowledge base containing logic facts shared among collocated devices. Unlike context-aware tuples, rules in the Fact Space Model are not exchanged between collocated devices and are not bound to facts to limit the perception of context information.

Scoping Mechanisms. A number of approaches support scoping mechanisms in the context of tuple spaces. Coordination with Scopes [21] introduces the concepts of scope for a tuple space. A scope represents a view on a flat tuple space. A set of operations is defined on those views allowing scopes to be joined, nested, intersected and subtracted. Tuples may thus be visible from several different scopes. This mechanism is mainly used to structure tuple spaces according to different viewpoints on a flat tuple space. However, scopes do not limit the propagation of tuples, i.e., tuples are propagated to other tuple spaces but may not be visible for certain scopes. Since the system was not devised for mobile computing applications, they rely on a centralized infrastructure. In contrast, TOTAM does not rely on any fixed infrastructure and tuples can be propagated through spontaneously formed MANETs.

CAMA [22] is an agent-based tuple space system which allows the definition of a scope which agents can join and leave. Scopes are defined as containers in a tuple space and can be nested in order to form hierarchical structures. This notion of scope improves on coordination with scopes since inserted tuples are only transmitted to agents which reside in the same scope. However, in order to send tuples to other scopes the agent first needs to change its scope. Tuples which are inserted in a specific scope can not be propagated automatically to other scopes. In TOTAM, by allowing the tuple itself to decide whether it should be propagated, more fine-grained sharing strategies can be expressed.

L2imbo [3] is a tuple-space based platform for mobile computing which provides special features for quality of service. Similar to CAMA, L2imbo introduces the concept of multiple tuple spaces but suffers from the same limitations as tuples do not have the ability to decide to which tuple space they should be propagated. It is interesting to note that L2imbo supports time-outs associated with tuples. This makes possible for the system to reorder tuples to make optimal use of the available network connectivity. Similar to CAMA, L2imbo introduces the concept of multiple tuple spaces but suffers from the same limitations as tuples do not have the ability to change to which tuple space they should be propagated.

Evolving tuples [23] have a field destination that they can change while they are hopping. This destination field is used to determine where the tuple will be transmitted

to after leaving a host. However, the destination field can only be a broadcast address or a specific host address. In order to send the tuple to two broadcast addresses the programmer will have to read the tuple and reinsert it to another broadcast address. Our approach uses semantic information to determine where it can be transmitted thus allowing more fine-grained propagation rules.

Inspired by LIME, TeenyLIME [24] introduces abstractions specially designed for wireless sensor networks (WSN). Every tuple space in TeenyLIME is shared only with one-hop neighbours, limiting the scope of tuples to one hop. Such limitation fits natural with WSN architectures where every node typically needs access to nearby information [24]. Scoping was introduced to address the scarce resources issue in the context of WSN. However, no other means are provided to express different propagation protocols, which need to be expressed in terms of single-hop operations.

Leasing Models. Garbage collection of unused tuples is a known problem in tuple space approaches [25]. Typically, a tuple space stores tuples which may never be subject to a remove operation, and which may never be garbage collected. To solve this problem, TOTAM employs the notion of leasing. Some traditional tuple space approaches such as Objective Linda [26], JavaSpaces [27] and TSpaces [28], augmented Linda's operations with a timeout: if a matching tuple is not found within the timeout, the operation returns an error. In JavaSpaces and TSpaces, a tuple can be inserted in the tuple with a *lease time* denoting the maximum amount of time before the tuple is automatically removed from the tuple space. However, the centralized nature of those approaches does not make them applicable in a mobile environment.

To the best of our knowledge, Tiamat [29] is the only tuple space model for the mobile environment that includes a leasing model. Tiamat follows a federated tuple space model in which each operation is leased. Interestingly, the authors describe that in Tiamat "leases may be based on time or on other measures such as the number of remote instances contacted". Unfortunately, no code examples are provided to see how programmers can declare leases based on such conditions. Tiamat leasing model incorporates the concept of expiration in the tuple space operations, but it does not provide listeners which allows application to react to them.

In [25], Mamei et Zambonelli remark the importance of incorporating a garbage collection mechanism to TOTA to remove unused tuples. In TOTA, one can encode a leased tuple such as the one described in our work by means of a custom propagation rule. In particular, the propagation rule needs to update its content to take into account the notion of time, and stop its propagation. The `MessageTuple` class described in the latest version of TOTA [25] defines a tuple that floods the network and deletes itself after some time has passed. In TOTAM, because leasing is orthogonal to the propagation protocol, programmers do not need to manually encode in the propagation rule the passage of time, nor the retraction of the tuple from the network upon a `delete` operation.

In [30], Ommici et al. extend ReSpecT [31], logic-based tuple space language with the notion of time. ReSpecT tuple centers behave like tuple spaces whose behaviour is specified in terms of reactions to events occurring in the tuple space. The notion of time is introduced into a tuple center with (1) some temporal predicates to get information about tuple center and event time, and (2) timed reactions which specify reactions

triggered by time events. Based on this extension, they discuss how some abstractions that could be built by introducing leasing into tuples similar to what TOTAM supports. An interesting topic of future work would be to investigate whether introducing leasing in TOTAM along the principles of (timed) tuple centers will allow us to build time-based coordination patterns in a more modular way than our current solution based on propagation protocols.

Publish/subscribe Systems. Many researchers have proposed publish/subscribe as a suitable communication model for MANETs because of its loosely coupled nature [32, 33, 18, 34]. Publish/subscribe middleware typically does not offer abstractions for representing failures or reacting to network connectivity. In fact, communicating parties are usually not aware of the underlying network configuration, or even if a published event was received by any subscriber; either failures are transparent to publishers and subscribers and the event notification service buffers events when subscribers disconnect, or the publishers are responsible themselves for encoding failure handling and events get lost if subscribers move out of communication range. For example, STEAM [35] allows publish/subscribe based on physically location, but it is hard to describe scopes based on semantic information as shown in the ambient game.

With respect to our support for context-awareness, context-aware publish subscribe (CAPS) [36] is the closest work as it allows certain events to be filtered depending on the context of the receiver. More concretely, the publisher can associate an event with a *context of relevance*. However, CAPS is significantly different from context-aware tuples. First, CAPS does not allow reactions on the removal of events, i.e., there is no dedicated operation to react when an event moves out of context. Moreover, it does not provide coordination of distributed parties, i.e., atomic removal of events is not supported. And last, the context of relevance is always associated to a physical space.

Some publish subscribe systems have explored the concept of scope for events. In scoped REBECA [37] systems can create a scope in which events will be published. A scope can be extended and therefore form a tree of scopes. Subscribers will only receive events of publishers which are in the same scope or have a common ancestor in the scope hierarchy. Similar to CAMA, publishing an event in an other scope requires the publisher to change it scope first, forcing developers to manually reinsert the tuples into the other scopes. Location-based publish/subscribe [38] suffers from the same limitation.

11. Availability

As previously mentioned, TOTAM has been implemented in AmbientTalk, and its implementation is shipped with AmbientTalk's standard language library[10]. In order to use the TOTAM middleware, developers need to import the TOTAM module accessible via the `lobby.at.lang.totam` namespace. This module provides the core model including the scoping mechanism based on propagation protocol and the leasing model.

[10]AmbientTalk's standard library can be downloaded with the language at `http://soft.vub.ac.be/amop`

The context rules that can be attached to a tuple have been implemented as a separated module extending TOTAM that is accessible via the `lobby.frameworks.tuples` namespace. The code for the three described applications (Guanotes, weScribble and Flikken) is also accessible within the AmbientTalk's standard language library.

The rule engine in TOTAM incorporates a truth maintenance system built on top of a RETE network[39]. A RETE network optimizes the matching phase of the inference engine providing an efficient derivation of context rule activation and deactivation. The network has also been optimized to allow constant time deletions by applying a *scaffolding* technique[40]. For details about the engine and its performance, we refer to[41].

A TOTAM system relies on AmbientTalk's service discovery facilities (based on multicast using UDP) to discover other systems in the network. Flikken's initial experimental setup was a set of HTC P3650 Touch Cruises phones running on J2ME (CDC) and communicating by means of TCP broadcasting on a wireless ad hoc WiFi network. Guanotes and weScribble's experimental setup is, however, a combination of Samsung Nexus S and Galaxy Nexus phones (running Android 4.0.4) and Motorola Xoom tablets (running Android 3.2). This is today the primary experimental setup for TOTAM's applications. As such it remains future work to port Flikken's current Java AWT GUI to the Android platform.

12. Conclusion

In this paper, we proposed a novel tuple space model specially designed for the development of mobile context-aware applications. TOTAM combines ideas from both federated and replication-based tuple spaces into a consistent tuple space-based framework. Unlike existing tuple space approaches, *only* the subset of tuples which should be perceivable, is made accessible to applications in TOTAM. This is achieved by extending tuples with a predicate, called a *context rule*, which determines when the receiving application is in the right context to perceive the tuple.

There are three main novelties to our approach. First, the use of context rules combined with the introduction of a rule engine in the tuple space system. This gives the programmer control over which tuples present in the tuple space should be actually accessible by applications. Programmers do not longer need to infer tuple perception manually as the rule engine takes care of it in an efficient way, making the code easier to understand and maintain. Second, TOTAM integrates the concept of *leasing* into a replicated-based tuple space model. Leasing is an essential concept in order to deal with permanent failures in a highly disconnected network topology. Our leasing abstractions allow the developer to determine upper boundaries on the availability of the injected tuples in the system independent of the connectivity. And finally, TOTAM introduces the concept of *antituples* to enable the "unsending" tuples injected to the network. Antituples are injected into the TOTAM network upon removal or retraction operations, avoiding programmers to manually encode them in terms of propagation protocols.

We have given both an informal and a formal account of the core features of TOTAM. The applicability of our model has been demonstrated by showing the implementation of *Flikken*, a mobile game in which players equipped with mobile devices

interact in a physical environment augmented with virtual objects, and *weScribble*, a collaborative peer-to-peer drawing application. Moreover, our prototype implementation of TOTAM has been used in an industrial case with the STIB and is actively used for teaching students at the Vrije Universiteit Brussel.

Acknowledgments

We would like to thank the STIB for providing us with the necessary resources, and the anonymous reviewers for their helpful comments.

References

[1] A. Murphy, G. Picco, G.-C. Roman, LIME: A middleware for physical and logical mobility, in: Proceedings of the The 21st International Conference on Distributed Computing Systems, IEEE Computer Society, 2001, pp. 524–536.

[2] C. Mascolo, L.Capra, W. Emmerich, Mobile Computing Middleware, in: Advanced lectures on networking, Springer-Verlag, 2002, pp. 20–58.

[3] N. Davies, A. Friday, S. Wade, G. Blair, L2imbo: a distributed systems platform for mobile computing, Mob. Netw. Appl. 3 (1998) 143–156.

[4] C. Julien, G.-C. Roman, Active coordination in ad hoc networks, in: R. D. Nicola, G. L. Ferrari, G. Meredith (Eds.), Coordination Models and Languages, 6th International Conference, COORDINATION 2004, Pisa, Italy, February 24-27, 2004, Proceedings, volume 2949 of *Lecture Notes in Computer Science*, Springer, 2004, pp. 199–215.

[5] M. Mamei, F. Zambonelli, Programming pervasive and mobile computing applications with the TOTA middleware, in: PERCOM '04: Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications, IEEE Computer Society, Washington, DC, USA, 2004, p. 263.

[6] T. Van Cutsem, S. Mostinckx, E. Gonzalez Boix, J. Dedecker, W. De Meuter, Ambienttalk: object-oriented event-driven programming in mobile ad hoc networks, in: Inter. Conf. of the Chilean Computer Science Society (SCCC), IEEE Computer Society, 2007, pp. 3–12.

[7] C. Scholliers, E. Gonzalez Boix, W. De Meuter, T. D'Hondt, Context-aware tuples for the ambient, in: On the Move to Meaningful Internet Systems, OTM 2010, Springer, 2010, pp. 745–763.

[8] D. Gelernter, Generative communication in Linda, ACM Transactions on Programming Languages and Systems 7 (1985) 80–112.

[9] A. Murphy, G. Picco, Using lime to support replication for availability in mobile ad hoc networks, in: 8th International Conference on Coordination Models and Languages (COORDINATION), LNCS, Springer-Verlag, 2006, pp. 194–211.

[10] P. Costanza, R. Hirschfeld, Language constructs for context-oriented programming: an overview of contextl, in: Proceedings of the 2005 symposium on Dynamic languages, DLS '05, ACM, New York, NY, USA, 2005, pp. 1–10.

[11] S. Gilbert, N. Lynch, Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services, SIGACT News 33 (2002) 51–59.

[12] M. Viroli, M. Casadei, Biochemical tuple spaces for self-organising coordination, in: COORDINATION '09: Proceedings of the 11th International Conference on Coordination Models and Languages, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 143–162.

[13] M. Viroli, A. Omicini, Coordination as a service, Fundamenta Informaticae 73 (2006) 507–534.

[14] E. Gonzalez Boix, A. Lombide Carreton, C. Scholliers, T. Van Cutsem, W. De Meuter, T. D'Hondt, Flocks: Enabling Dynamic Group Interactions in Mobile Social Networking Applications, in: Proceedings of the 2011 ACM Symposium on Applied Computing (SAC), Taichung, Taiwan, March 21–25, 2011, volume 1, ACM, 2011, pp. 425–432.

[15] A. S. Tanenbaum, M. V. Steen, Distributed Systems: Principles and Paradigms, Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.

[16] C. Scholliers, E. Gonzalez Boix, W. De Meuter, Totam: Scoped tuples for the ambient, in: Proc. of the CAMPUS Workshop collocated with DisCoTec'09 federated event, volume 19, EASST, 2009, pp. 19–34.

[17] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, W. De Meuter, Ambient-oriented Programming in Ambienttalk, in: D. Thomas (Ed.), Proceedings of the 20th European Conference on Object-oriented Programming (ECOOP), volume 4067 of *Lecture Notes in Computer Science*, Springer, 2006, pp. 230–254.

[18] P. T. Eugster, P. A. Felber, R. Guerraoui, A.Kermarrec, The many faces of publish/subscribe, ACM Computing Survey 35 (2003) 114–131.

[19] J. Pauty, P. Couderc, M. Banatre, Y. Berbers, Geo-linda: a geometry aware distributed tuple space, in: AINA '07: Proceedings of the 21st International Conference on Advanced Networking and Applications, IEEE Computer Society, Washington, DC, USA, 2007, pp. 370–377.

[20] S. Mostinckx, C. Schlliers, E. Philips, C. Herzeel, W. De Meuter, Fact spaces: Coordination in the face of disconnection, in: Proceedings of 9th International Conference on Coordination Models and Languages, volume 4467 of *Lecture Notes in Computer Science*, Springer-Verlag, Heidelberg, 2007, pp. 268–285.

[21] I. Merrick, A. Wood, Coordination with scopes, in: SAC '00: Proceedings of the 2000 ACM symposium on Applied computing, ACM, New York, NY, USA, 2000, pp. 210–217.

[22] A. Iliasov, R. A., Structured coordination spaces for fault tolerant mobile agents, Advanced Topics in Exception Handling Techniques 4119 (2006) 181–199.

[23] D. Stovall, C. Julien, Resource discovery with evolving tuples, in: ESSPE '07: International workshop on Engineering of software services for pervasive environments, ACM, New York, NY, USA, 2007, pp. 1–10.

[24] P. Costa, L. Mottola, A. Murphy, G. Picco, Teenylime: transiently shared tuple space middleware for wireless sensor networks, in: MidSens '06: Proceedings of the international workshop on Middleware for sensor networks, ACM, New York, NY, USA, 2006, pp. 43–48.

[25] M. Mamei, F. Zambonelli, Programming pervasive and mobile computing applications: The tota approach, ACM Trans. Softw. Eng. Methodol. 18 (2009) 15:1–15:56.

[26] T. Kielmann, Designing a coordination model for open systems, in: Proceedings of the First International Conference on Coordination Languages and Models, COORDINATION '96, Springer-Verlag, London, UK, UK, 1996, pp. 267–284.

[27] E. Freeman, K. Arnold, S. Hupfer, JavaSpaces Principles, Patterns, and Practice, Addison-Wesley Longman Ltd., Essex, UK, 1999.

[28] P. Wyckoff, S. W. McLaughry, T. J. Lehman, D. A. Ford, T spaces, IBM Systems Journal 37 (1998) 454–474.

[29] G. P. McSorley, H. Evans, Tiamat: Generative communication in a changing world., in: Middleware Workshops, PUC-Rio, 2003, pp. 37–44.

[30] A. Omicini, A. Ricci, M. Viroli, Time-aware coordination in respect, in: Proceedings of the 7th international conference on Coordination Models and Languages, COORDINATION'05, Springer-Verlag, Berlin, Heidelberg, 2005, pp. 268–282.

[31] E. Denti, A. Natali, A. Omicini, On the expressive power of a language for programming coordination media, in: Proceedings of the 1998 ACM symposium on Applied Computing, SAC '98, ACM, New York, NY, USA, 1998, pp. 169–177.

[32] R. Meier, Communication paradigms for mobile computing, SIGMOBILE Mob. Comput. Commun. Rev. 6 (2002) 56–58.

[33] G. Cugola, H.-A. Jacobsen, Using publish/subscribe middleware for mobile systems, SIGMOBILE Mob. Comput. Commun. Rev. 6 (2002) 25–33.

[34] Y. Huang, H. Garcia-Molina, Publish/subscribe in a mobile environment, Wirel. Netw. 10 (2004) 643–652.

[35] R. Meier, V. Cahill, Exploiting proximity in event-based middleware for collaborative mobile applications, in: Distributed Applications and Interoperable Systems.

[36] D. Frey, G.-C. Roman, Context-aware publish subscribe in mobile ad hoc networks, in: 9th International Conference on Coordination Models and Languages (COORDINATION), volume 4467 of *LNCS*, Springer, 2007, pp. 37–55.

[37] F. Ludger, M. Mezini, G. Mühl, A. Buchmann, Engineering event-based systems with scopes, in: ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming, Springer-Verlag, London, UK, 2002, pp. 309–333.

[38] P. Eugster, B. Garbinato, A. Holzer, Location-based publish/subscribe, Fourth IEEE International Symposium on Network Computing and Applications (2005) 279–282.

[39] C. L. Forgy, Rete: A fast algorithm for the many pattern/many object pattern match problem, in: Artificial Intelligence & Databases, Kaufmann Publishers, INC., San Mateo, CA, 1989, pp. 547–557.

[40] M. Perlin, Scaffolding the RETE network, in: International Conference on Tools for Artificial Intelligence, IEEE Computer Society, 1990, pp. 378–385.

[41] C. Scholliers, E. Philips, Coordination in Volatile Networks, Master's thesis, Vrije Universiteit Brussels, 2007.

Appendix A. TOTAM's Programming API

Table A.4 summarizes TOTAM's programming API which is described in Section 5.

Middleware operations	
`makeTupleSpace(descriptor)`	creates a tuple space with the given descriptor.
`tuple:` `content` **`withPropagationProtocol:`** `protocol`	creates a tuple with the given list of fields. The optional parameter **`withPropagationProtocol:`** denotes the propagation protocol by which the tuple may be injected in the network.
`propagationProtocol:` `closure`	returns a protocol object which extends the default propagation protocol with the given code block.
`var:` `symbol`	return a variable from the given symbol.
Tuple space operations	
`goOnline`	publishes the tuple space in the TOTAM network.
`rdp(template)`	returns a tuple matching the template or **`nil`** if none is present at the time of invocation.
`rdg(template)`	returns all tuples matching the template or **`nil`** if none is present at the time of invocation.
`out(tuple, lease)`	adds a private tuple to the tuple space.
`inject:` `tuple` **`inContext:`** `rule` **`withLease:`** `interval`	adds a public tuple to the tuple space. **`inContext:`** and **`withLease:`** optional parameters allow to specify the associated context rule and (lease) time interval, respectively. It returns a publication object with methods to stop the tuple's propagation, and retract it from the network.
`when:` `template` **`read:`** `closure` **`outOfContext:`** `oocClosure`	registers a reaction on the tuple space for the given template. When a tuple matching the template is available in the tuple space, the `closure` listener is applied binding all variables of the template to the matching tuple. Optionally, the **`outOfContext:`** closure can be specified to react when the matching tuple is no longer perceivable.
`when:` `template` **`in:`** `closure` **`outOfContext:`** `oocClosure`	works analogously to **`when:read:outOfContext:`** operation, but registers a reaction on the removal of a tuple matching the template.
`whenever:` `template` **`read:`** `closure` **`outOfContext:`** `oocClosure`	works analogously to **`when:read:outOfContext:`** operation, but triggers the `closure` listener for *every* perceivable tuple matching the template.
`whenever:` `template` **`in:`** `closure` **`outOfContext:`** `oocClosure`	works analogously to **`when:in:outOfContext:`** operation, but triggers the `closure` listener for *every* perceivable tuple matching the template.

Table A.4: Programming API of TOTAM