Chapter 1
Tuple Space Middleware for Wireless Networks

Paolo Costa, Vrije Universiteit, Amsterdam, The Netherlands,
costa@cs.vu.nl
Luca Mottola, Politecnico di Milano, Italy
mottola@elet.polimi.it
Amy L. Murphy, FBK-IRST, Povo, Italy
murphy@fbk.eu
Gian Pietro Picco, University of Trento, Italy
gianpietro.picco@unitn.it

1.1 Introduction

Wireless networks define a very challenging scenario for the application programmer. Indeed, the fluidity inherent in the wireless media cannot be entirely masked at the communication layer: issues such as disconnection and a continuously changing execution context most often must be dealt with according to the application logic. Appropriate abstractions, usually provided as part of a *middleware*, are therefore required to support and simplify the programming task.

Coordination [1] is a programming paradigm whose goal is to separate the definition of the individual behavior of application components from the mechanics of their interaction. This goal is usually achieved by using either *message passing* or *data sharing* as a model for interaction. Publish-subscribe, described in Chapter **??** is an example of the former, where coordination occurs only through the exchange of messages (*events*) among publishers and subscribers. While message passing, in its pure form, is inherently *stateless*, data sharing enables coordination among components by manipulating the (distributed) *state* of the system. The tuple space abstraction, the subject of this chapter, is a typical example of a data sharing approach. The two models, publish-subscribe and tuple spaces, have sometimes crossed paths in the scientific literature: their expressive power has been compared on formal grounds in [2]; the limits of an implementation of a stateful tuple space on top of a stateless publish-subscribe layer has been investigated in [3]; some extensions of publish-subscribe with stateful features exist (e.g., the ability to query over past events as in [4]). A thorough discussion of the relationship between the two is outside the scope of this chapter, and hereafter we focus solely on tuple spaces.

Linda [5] is generally credited with bringing the tuple space abstraction to the attention of the programming community. In Linda, components communicate through a shared *tuple space*, a globally accessible, persistent, content-addressable data structure containing elementary data structures called *tuples*. Each tuple is a sequence of typed fields, as in $\langle$"foo", 9, 27.5$\rangle$, containing the information being communicated. A tuple t is inserted in a tuple space through an **out**(t) oper-

ation, and can be withdrawn using **in**(p). Tuples are anonymous, their selection taking place through pattern matching on the tuple content. The argument p is often called a *template* or *pattern*, and its fields contain either *actuals* or *formals*. Actuals are values; the fields of the previous tuple are all actuals, while the last two fields of ⟨"foo", ?integer, ?float⟩ are formals. Formals act like "wild cards", and are matched against actuals when selecting a tuple from the tuple space. For instance, the template above matches the tuple defined earlier. If multiple tuples match a template, the one returned by **in** is selected non-deterministically. Tuples can also be read from the tuple space using the non-destructive **rd**(p) operation. Both **in** and **rd** are blocking, i.e., if no matching tuple is available in the tuple space the process performing the operation is suspended until a matching tuple becomes available. The asynchronous alternatives **inp** and **rdp**, called *probes*, have been later introduced to allow the control flow to return immediately to the caller with an empty result when a matching tuple is not found. Moreover, some Linda variants (e.g., [6]) also provide *bulk operations*, **ing** and **rdg**, used to retrieve all matching tuples in one step.

The fact that only a small set of operations is necessary to manipulate the tuple space, and therefore to enable distributed component interaction, is per se a nice characteristic of the model. However, other features are particularly useful in a wireless environment. In particular, coordination among processes in Linda is decoupled in time and space, i.e., tuples can be exchanged among producers and consumers without being simultaneously available, and without mutual knowledge of their identity or location. This decoupling is fundamental in the presence of wireless connectivity, as the parties involved in communication change frequently due to migration or fluctuating connectivity patterns[1]. Moreover, tuple spaces can be straightforwardly used to represent the context perceived by the coordinating components. On the other hand, this beneficial decoupling is achieved thanks to properties of the Linda tuple space—its global accessibility to all components and its persistence—difficult to maintain in a dynamic environment with only wireless links.

In the last decade, a number of approaches were proposed that leverage the beneficial decoupling provided by tuple spaces in a wireless setting, while addressing effectively the limitations of the original Linda model. Our group was among the first to recognize and seize the potential of tuple spaces in this respect, through the LIME model and middleware [8]. This chapter looks back at almost a decade of efforts in the research community, by concisely describing some of the most representative systems and analyzing them along some fundamental dimensions of comparison. In doing so, it considers two main classes of applications that rely on wireless communication. First, Section 1.2 considers *mobile networks*, where the network topology is continuously redefined by the movement of mobile hosts. Then, Section 1.3 considers the more recent scenario defined by *wireless sensor networks* (WSNs), networks of tiny, resource-scarce wireless devices equipped with sensors and/or actuators, enabling untethered monitoring and control.

The structure of these two sections is identical. Each first provides a brief survey of representative systems in the corresponding class of wireless applications. Then,

[1] A rather abstract treatment of coordination and mobility can be found in [7].

it elicits some recurring themes and dimensions of comparison, which are used to classify and compare the systems. Finally, a small case study is presented to show how tuple spaces can be used in the context of a realistic application for the wireless domain at hand. The case studies are borrowed from the work of the authors, and are based respectively on LIME [9] and its adaptation to WSNs, TeenyLIME [10].

Finally, Section 1.4 offers some concluding remarks.

1.2 Mobile Networks

This section discusses the applicability of tuple spaces to environments with moving hosts. It considers two primary mobility models, namely nomadic and mobile ad hoc networks (MANETs). In nomadic mobility, a wired infrastructure supports the connection of mobile devices to the wired network through base stations. Instead, MANET removes the infrastructure and nodes communicate directly only when they are within range.

1.2.1 Representative Systems

The following outlines the main tuple space approaches developed for mobile settings. Due to space constraints, this should not be considered an exhaustive list, but rather an outline of representative systems. The sequence of presentation roughly corresponds to the chronological order of appearance of each system.

TSpaces. After the initial enthusiasm in the 1980s with tuple spaces, the 1990s saw a resurgence of the model for distributed computing. IBM's TSpaces [11] was one of the first such systems, providing a client/server interface to a centralized tuple space, merging the simplicity and flexibility of the tuple space front-end with an efficient database back-end. TSpaces targets both fixed, distributed systems, as well as provides support for nomadic computing with devices such as hand held PDAs. In both cases, all data is centrally managed and clients remotely issue operations.

L^{2}imbo. While TSpaces relies on a centralized server, the L^{2}imbo platform [12] proposes a decentralized implementation in which each host holds a replica of the tuple space. The underlying mobility model is still nomadic, forcing mobile hosts to connect through a base station to gain full access to the data. Nevertheless, because the tuple space is replicated locally, hosts can perform limited operations even during disconnections.

The implementation leverages off IP multicast to disseminate updates and ensure consistency among replicas. Tuple spaces are uniquely mapped to multicast groups and all interested hosts must be members. When a host joins a group, a local replica of the tuple space is created and all subsequent updates are received as multicast messages. To avoid conflicts, the authors introduce the notion of *tuple ownership*.

While reading operations are always permitted, writing operations can be performed only after the ownership for the specific tuple has been acquired.

In L^2imbo all tuples are associated with a type, which is used, along with traditional field-based matching, by the **in** and **rd** requests to retrieve the desired tuples. To enable fine-grained searches, types are arranged in a hierarchy and a match occurs if a tuple of the type or any subtype is found.

LIME. The LIME model [9] defines a coordination layer that adapts and extends the Linda model towards applications that exhibit physical mobility of hosts in a MANET and/or logical mobility of software agents [13]. Given the topic of this chapter, hereafter we bias our presentation towards the former case. LIME, a mobile host has access to a so-called interface tuple space (ITS), permanently and exclusively attached to the host itself. The ITS, accessed using Linda primitives, contains tuples that are physically co-located with the host and defines the only data available to a lone host. Nevertheless, this tuple space is also transiently shared with the ITSs belonging to the mobile hosts currently within communication range. When a new host arrives, the tuples in its ITS are conceptually merged with those already shared, belonging to the other mobile hosts, and the result is made accessible through the ITS of each of the hosts. This provides a mobile host with the illusion of a local tuple space containing tuples coming from all the hosts currently accessible, without any need to know them explicitly.

LIME also augments the Linda model with the notion of *reaction*. A reaction $\mathscr{R}(s,p)$ is defined by a code fragment s specifying the actions to be performed locally when a tuple matching the pattern p is found in the shared tuple space. This effectively combines the proactive style, typical of tuple space interaction, with the reactive paradigm useful in the dynamic, mobile environment. Recent papers further enhance the LIME model by adding support for security [14], replication [15] and code deployment [16].

EgoSpaces. EgoSpaces [17] is a tuple space middleware similar to LIME exploiting a fully distributed architecture. In its model, the network is perceived as an underlying database of tuples. Each host defines its own tuple space by creating a *view*, i.e., a subset of the tuples available at other hosts, selected according to the specified constraints (e.g., host IDs, number of hops, tuple patterns).

Hosts interact with these views through the basic Linda operations and special constructs for event-driven communication. EgoSpaces also provides transactional support to ensure that a sequence of operations (e.g., a **rd** followed by an **in**) is executed atomically.

TOTA. In contrast to the previously described systems, TOTA [18] exploits a tuple-centric approach to support both pervasive computing and MANET mobility. In TOTA, tuples spread hop-by-hop among nodes according to the rules specified in the tuple itself. These rules may include, for example, the scope of the tuple (i.e., how many hops the tuple should travel) or the conditions for the propagation to occur (e.g, if a tuple denotes a fire alarm, it will be replicated only if the room temperature is above a threshold). Tuples can also be modified along the way to take into account changing conditions (e.g., a tuple containing the temperature can average its value

	TSpaces [11]	**L^{2}imbo** [12]	**LIME** [9]	**EgoSpaces** [17]	**TOTA** [18]
Architecture	Centralized	Decentralized	Decentralized	Decentralized	Decentralized
Mobility scenario	Nomadic	Nomadic	MANET	MANET	MANET
Context-awareness	N/A	QoS attributes	Context as data	Context as data	Context as data
Disconnected operation	None	Read-only	Yes	Yes	None
Atomicity	Yes (strong)	No	Yes (strong)	Yes (best-effort)	No
Reactions (to)	Yes (operations)	No	Yes (state)	Yes (state)	Yes (operations)
Scope	Whole TS	Whole TS	Federated TS or single host/agent	Programmer-defined views	Local or one hop
Other extensions	Probes, bulk, time-outs	Time-outs	Probes, bulk	Probes, bulk	Bulk

Table 1.1 Features of representative tuple space systems for mobile computing.

over all the readings found). Hosts can access the local or neighboring tuple spaces to retrieve, add, or remove tuples. Event-driven constructs are also present to react when a particular tuple is inserted in the tuple space.

1.2.2 Discussion

We now focus on the features that distinguish the above representative systems. Table 1.1 provides a concise summary while details are provided below.

Architecture and Mobility Scenario. The placement of the tuple space data plays a key role in the applicability of the model, especially in relation to the target mobility scenario. For example, the centralized, client-server model of TSpaces supports nomadic mobility for resource poor devices such as PDAs. It also allows the server to both take on the majority of the computation burden as well as provide optimizations to data access, e.g., with a database back-end, which is not possible on small, mobile devices.

Instead, to support MANETs, a single server will not always be reachable and each node essentially becomes a mini-server with part of the global data. This is the decentralized model adopted by TOTA as well as the transiently shared tuple spaces of LIME and EgoSpaces. While this fully distributed model requires direct interaction among nodes to share locally hosted data, at the same time it eliminates any central point of coordination, an essential requirement in the MANET environment.

L^{2}imbo strikes a balance between the two approaches, supporting nomadic mobility with decentralized tuple spaces. The primary advantage is support for disconnected operation, as discussed later.

Context-Awareness. As outlined in [19], tuple spaces naturally support the requirements of context-aware applications. Indeed, context data can be stored in the tuple space just as any other data, as proposed by LIME, EgoSpaces and TOTA, leveraging the decoupled nature of Linda to separate the context data producers from the consumers.

L^{2}imbo takes this one step further by exposing context information critical to supporting quality of service in the mobile environment. This is accomplished

through a set of monitors running on each host. *Connectivity* monitors check the connectivity between a pair of hosts and report quality (e.g., throughput). *Power* monitors, instead, observe the power level on the local host as well as on nearby hosts such that applications can employ an appropriate saving scheme. Finally, *cost* monitors track the communication costs between two hosts. Based on this information, applications can monitor local and remote resources and, if necessary, activate specific energy-saving policies.

Despite the suitability of the tuple space abstraction, the standard Linda matching based on types and exact values is often insufficient for context-aware applications as such queries frequently require range rather than exact value matching. For example, a query may look for any host within a 50 m radius of its current location. This has been addressed in LIGHTS [20], the tuple space underlying LIME. To support the needs of context-awareness, LIGHTS extends Linda matching semantics to use range matching, fuzzy logic comparison operators, and other extensions enhancing the expressiveness of tuple queries.

Disconnected Operation. Given the dynamicity of the mobile scenario, connectivity cannot be guaranteed at all times. In some systems such as TSpaces, disconnections are considered a fault, and disconnected clients cannot access the tuple space. Instead other systems adopt the view that disconnection is an expected event in mobile computing, and some amount of functionality must be provided even when a node is isolated from all others.

L^2imbo tolerates disconnected operations by keeping a replica of the entire tuple space on each host. Eventual consistency among different copies is ensured by a variant of IP multicast [21] in which each tuple space is bound to a multicast group to which every node accessing the tuple space belongs. While disconnected, a host can optimistically read tuples from its local cache, but it cannot remove any tuples. When a host reconnects to the network, it communicates to the multicast group the tuples created during the disconnection and receives notification of all tuples added/removed by other nodes.

In the MANET environment, LIME allows a node to operate freely on any tuples in its federated tuple space. When isolated, a node has access only to the tuples it hosts, but, still, any operations are possible. An extension [15] addresses replication at the tuple level based on patterns. This gives the individual agent control over the amount of data replicated and hence transmitted across the wireless links. The update policy is also flexible, allowing updates to propagate only from the master, from any newer version, or never.

Atomicity. A key feature of any modern distributed platform is the ability to perform multiple operations on a diverse set of hosts in a single atomic step. This is particularly true for the stateful model of tuple spaces because the lack of atomicity may lead to an inconsistent system state, e.g., when two hosts succeed in removing the same tuple.

To this end, mainstream systems such as TSpaces offer native transactional support by using a database server as a backend. Unfortunately, this approach is overly complex for systems of mobile devices with fluctuating connections. To overcome

this, LIME exploits transactions only to manage group membership by implicitly electing a single leader for each group, responsible for ensuring the atomicity of tuple space *engagement* and *disengagement*, respectively the process of merging or breaking down the tuple space based on a change in connectivity. Conversely, atomicity for distributed operations (e.g., **in**) are guaranteed through a lightweight approach, relying on a combination of reactions and non-blocking probes. When an **in** operation is issued, if a matching tuple is found in the local tuple space, the operation immediately returns. Otherwise, the run-time installs a reaction for the same pattern. When this reactions fires, a non-blocking **inp** is issued to remove the tuple. If, in the meanwhile, the tuple has been withdrawn by another host, the reaction remains in place and the process continues to wait.

One drawback of the LIME approach, however, is that atomicity can be guaranteed only for system-supported operations. To address this concern, EgoSpaces proposes a traditional, yet costly, transaction-based mechanism. In addition, it provides a best-effort solution based on *scattered probes*. These operations are implemented such that all the hosts participating in the view are contacted one at a time and an empty set is returned if no matching tuple has been found. Scattered probes provide a weaker consistency because they are allowed to miss a matching tuple in the view. On the other hand, the implementation is both more efficient because transactions are not employed and more flexible because hosts are not suspended.

Reactions. Mobility challenges many of the assumptions made in traditional distributed computing. Of primary concern is that data is only transiently accessible due to changing connectivity among participants. This leads naturally to the introduction of some notion of reactive operation where, similar to event-based programming, a component can be notified when something interesting happens. In TSpaces and TOTA, nodes can be notified when certain *operations* (e.g., the insertion or removal of a tuple) are issued on the tuple space. However, the notification occurs only if the recipient is connected to the tuple space where the operation is issued. In LIME and EgoSpaces, instead, hosts are notified when data matching a given pattern appears in the host's federated tuple space. These *state-based* reactions are important in the MANET environment because two agents may not be connected when a tuple is inserted: the state-based semantics ensure that the reaction fires whenever a "new" relevant tuple is detected (e.g., when two nodes meet), and not just upon an insertion operation. Thus, being notified when data is *accessible* as opposed to when it is *inserted* provides a natural and very powerful programming primitive, useful in many mobile applications.

Scope. In large-scale mobile networks it is not practical to share the entire tuple space across all nodes. The overhead to route requests and replies would quickly drain all host resources. Therefore, most implementations limit the distribution to a single (physical) hop. TOTA introduces explicitly scoped operations, such as **readOneHop**, while LIME allows the programmer to set the scope of an operation to a single, specific host.

EgoSpaces [17], instead, enables the programmer to flexibly define the *scope* of the shared tuple space, called a *view*. This is expressed through a *declarative*

specification providing constraints over the properties of the underlying network (e.g., only nodes within 5 hops or 500 meters), on the host (e.g., only PDAs), and on the data (e.g., only location information).

Other Extensions. In addition to reactive operations and scoping, the transient accessibility of data changes the programming paradigm. In standard Linda programs for parallel computation, blocking operations are natural because, unless a process has data to work with, it should be suspended. In the mobile environment, instead, processes are often interactive or at least more flexible to the current context. Thus, if some needed data is not available, it may be better to switch to another task rather than block. Moreover, knowledge about the lack of data, which may also imply the unavailability of a host known to carry such data, may be important per se.

This kind of interaction is not supported by traditional blocking operations, thus most of the systems discussed in this section provide the non-blocking probes mentioned in Section 1.1, which query a tuple space and return either a matching tuple or a keyword indicating that no match exists. Such immediate return after the operation is issued gives the programmer a high degree of flexibility. An intermediate approach is taken in TSpaces and L^2imbo, where a timeout can be specified for (blocking) **in** and **rd** operations to avoid locking the process indefinitely.

Additionally, some applications logically create multiple versions of the same piece of information with each successive version invalidating the previous. For example, a location tuple constantly changes as a host moves through space. Each new location represents an update, replacing the now-irrelevant previous location. In most tuple space systems, it is only possible to remove the old data and insert new data, losing any logical connection between the two. Instead, it is meaningful to allow the data to be *changed*, associating it with the old data and at the same time identifying that it has been updated. Such a mechanism is provided by a variant of LIME [15] in which the user specifies a template for the old data together with the actual new data. The new data is distinguished from the old with a version number. This mechanism also serves as a building block upon which consistency between master and replicas is managed.

1.2.3 Tuple Spaces in Action

To illustrate concretely the benefits of the tuple space abstraction in the mobile environment, this section illustrates TULING [19], a sample application built on top of LIME for collaborative exploration of a space, emphasizing the exchange of context information. While offering a simple example, TULING demonstrates how the operations available in LIME are both natural and sufficient to provide the range of interaction necessary for exploiting context. Moreover, the interaction patterns of TULING are applicable to real-world collaborative applications, such as team coordination in a disaster recovery scenario.

Scenario and Requirements. TULING is intended to be used by multiple individuals moving through a common environment, each equipped with a GPS- and wireless-enabled PDA. Users see a representation of their current position as well as a trail representation of previous movements, as shown in Figure 1.1. When a new user comes within range, her name is displayed and one of several monitoring modes can be selected, e.g., retrieving only the current location, tracking the location as it changes, or retrieving the entire movement history of the user. The choice is typically based on the tolerance for the overhead associated to each type of monitoring.

TULING also allows users to add annotations, such as a textual note or a digital photograph, to their own current location. These annotations are indicated on the display with a special icon: by clicking on the icon, the annotation can be viewed as long as the requesting user is connected to the user who made the annotation.

Design and Implementation. The design and implementation of TULING focuses on making various aspects of the application, e.g., location, other users, and annotations, available as state inside the tuple space. This choice makes all data accessible to all connected hosts. In contrast, in a model such as publish-subscribe, explored in earlier chapters, data is typically transiently available only at the moment it is published, thus limiting the awareness of the data to the hosts that are connected. To overcome this restriction, a query-response paradigm can be exploited in parallel to proactively retrieve all missed data. In contrast, the LIME model of transiently shared tuple spaces provides a single interface for both modes of interaction, unifying the treatment of new and stored data as well as system-level information such as the presence of other hosts.

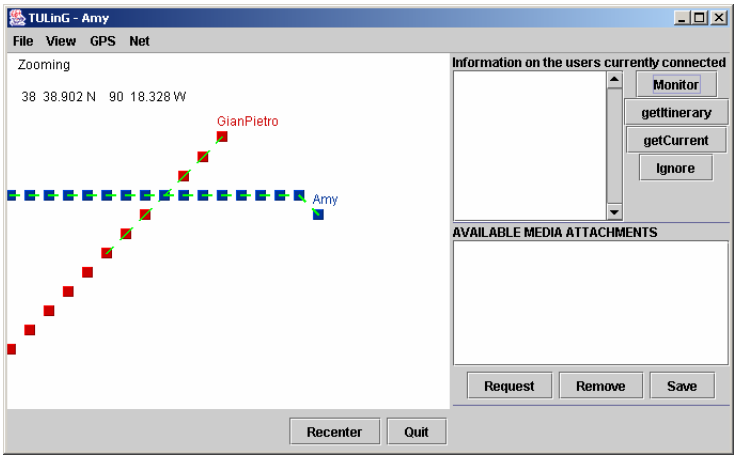

Fig. 1.1 Screenshot of a TULING user, Amy. While near a second user, GianPietro, his history and movement are visible, but once out of range, updates are no longer propagated and only the locally visible movements of Amy are displayed.

```
public class NewHostListener implements ReactionListener{
  NewHostListener() {
    LimeSystemTupleSpace lsts = new LimeSystemTupleSpace();
    // HOST arrival pattern
    Tuple newHostPattern = new Tuple().addActual("_host")
                                      .addFormal(LimeServerID.class);
    lsts.addListener({new LimeSystemReaction (newHostPattern, this,
                                              Reaction.ONCEPERTUPLE)});
  }
  void reactsTo(ReactionEvent re) {
    // display name of the new host arriving
    // the second field of re.getEventTuple() contains the host name
  }
}
```

Fig. 1.2 Code to react to the arrival of a new user using the `LimeSystemTupleSpace`.

The combined requirements to both monitor the current location of a user and to display the history information about the itinerary require that TULING provide access to the current and previous locations of a user. The *current location* is represented by a single tuple containing the GPS coordinates and a timestamp. To represent movement history, TULING uses a separate tuple template. The chosen solution groups multiple prior locations together into a single *stride* tuple that contains a sequence number and a list of locations. The number of locations in the stride list is tunable to balance the overhead of retrieving all the stride tuples to build a history against the overhead of updating the stride tuple with each new location. The updating of each type of location tuple is simply a matter of issuing an **out** followed by an **in**. This sequence ensures that at all times a location tuple is present in the tuple space. It is also worth noting that these operations are entirely local to the host where they are issued, therefore the overhead is minimal.

To access the location context of other hosts, TULING uses a combination of reactions and probe operations. Specifically, it uses a feature of LIME called the LimeSystem tuple space, a system-maintained tuple space that contains information about which hosts are currently connected. To display the name of a user within range, it employs a reaction on the LimeSystem tuple space. To give a feel for how simple it is to accomplish this operation, Figure 1.2 shows the required code to register a reaction for the arrival of a new host. A similar process is required for reacting to the departure of a host.

Once the name of a user is displayed, the monitoring mode must be selected. As a result of LIME's transient sharing of the tuple spaces among hosts within range, the locations of all connected users are available to one another. To get the current position, a non-blocking read operation **rdp**, restricted in scope to the selected host, is issued for the location tuple. A probing read is used to prevent the system from blocking in the case where the host disconnects before the read operation completes. Similarly, retrieving the itinerary requires a bulk read operation, **rdg**, on the stride tuples for the selected host. Instead, monitoring a user is accomplished by installing a reaction on the location tuple. Each time a new location is inserted, the reaction

fires remotely, a copy of the tuple is sent to the registered user, and their local display is updated. When two hosts disconnect causing their tuple spaces disengage and become no longer shared, the reaction ceases to fire. Nevertheless, when they come back within range, the reaction is automatically reinstalled by the system and updates propagate once again. It is worth noting that a similar reaction for location tuples scoped on the entire federated tuple space can be installed to display the locations of *all* users who come within range. In this case, the individual per-user reactions described above are unnecessary. The trade-off, however, is the inability to control the overhead as all users are unconditionally monitored.

The annotation feature is similarly supported with a combination of a reaction scoped over the entire federated tuple space to track which annotations are accessible, and a **rdp** to retrieve the contents of a requested annotation. Importantly, the implementation separates the knowledge of the existence of an annotation from the annotation itself, using two different tuple formats. This was motivated by the observation that the annotation contents themselves may be large, e.g., a digital photograph. Because a LIME reaction retrieves a copy of the matching tuple, reacting to the annotation contents would involve transferring the entire annotation whether or not it will be used, unnecessarily using the wireless communication media. Instead, by reacting to a small tuple which is essentially a reference to the actual annotation, the reaction processing remains efficient. The result, however, is the restriction that annotations can only be viewed while users are connected; a reasonable compromise for effectively managing overhead.

Experience with TULING clearly demonstrates the effectiveness of LIME to support context aware interactions in the MANET environment. The simple combination of key LIME operations, such as the probing **inp** and reactions, give the programmer rich mechanisms to interact with both the application and context data.

1.3 Wireless Sensor Networks

Wireless sensor networks (WSNs) pose peculiar challenges, only partially overlapping with those of mobile networks. Although communication occurs wirelessly, nodes tend to be static. Moreover, WSN devices typically offer much fewer resources than those employed in mobile networks. As a result, a good fraction of the programming effort often focuses on low-level concerns such as resource management. Abstractions such as tuple spaces can help programmers to address the requirements of WSN applications by raising the level of abstraction and hiding distribution.

Differently from the approaches described in the previous section, in WSNs few works provide a genuine tuple space abstraction to application programmers. Nevertheless, a relevant fraction of existing approaches leverages off the same first principles. The systems surveyed in this section indeed provide *data-centric* programming abstractions where the location of data, as well as the identity of the individual devices, plays only a secondary role. On top of this feature, they enable various forms

of *data sharing* among different devices. Consequently, distributed interactions occur implicitly in accessing some piece of data that programmers cannot a priori locate on some specific device. Blending these features in a single programming framework finds fertile ground in WSNs, where data is of paramount importance to application developers.

The structure of presentation is similar to the one we followed for mobile computing in the previous section. Section 1.3.1 describes exemplary approaches from the current state of the art. Again, the choice of systems to be discussed does not pretend to be exhaustive: the goal is to give the reader the insights necessary to appreciate how the driving concepts of tuple spaces have been applied in WSNs. Next, Section 1.3.2 illustrates the key features of the approaches described, comparing them against each other. Finally, Section 1.3.3 discusses a small case study to provide a concrete example of how tuple spaces are applicable to the WSN domain.

1.3.1 Representative Systems

This section surveys systems that either provide a tuple space abstraction to application developers, or more generally take inspiration from the key features of tuple spaces. In doing so, the discussion is limited to approaches geared towards programming individual WSN devices. Alternative paradigms have been explored in WSNs, whose goal is to give programmers a way to program the network as a whole. These approaches, commonly termed "macroprogramming" [22], radically depart from traditional programming. Therefore, they are not directly comparable with the ones discussed next.

Abstract Regions. Welsh et al. [23] propose a set of general-purpose communication primitives providing addressing, data sharing, and aggregation among a given subset of nodes. A *region* defines a neighborhood relationship between a specific node and other nodes in the system. For instance, a region can be defined to include all nodes within a given number of hops or within physical distance d. Data sharing is accomplished using a tuple space-like paradigm by giving developers language constructs to read/write $\langle key, value \rangle$ pairs at remote nodes. In a sense, this resembles the **rd** and **out** operations in traditional tuple space middleware, although the data format and matching is clearly much less expressive. Dedicated constructs are also provided to aggregate information stored at different nodes in a region. Moreover, a lightweight thread-like concurrency model, called Fibers, is provided for blocking operations. By their nature, Abstract Regions target applications exhibiting some form of spatial locality, e.g., tracking moving objects, or identifying the contours of physical regions.

Agilla. The work in [24] presents a middleware system for WSNs that adopts a mobile agent paradigm [13]. Programs are composed of one or more software agents able to migrate across nodes. In a sense, an Agilla agent is similar to a virtual machine with its own instruction set and dedicated data/instruction memory. Coordina-

tion among agents is accomplished using a tuple space. Agents insert data in a local data pool to be read by different agents at later times. The data of interest is identified using a pattern matching mechanism, in a way similar to what is described in Section 1.1. In Agilla, the use of tuple spaces allows one to decouple the application logic residing in the agents from their coordination and communication. At the same time, tuple spaces also provide a way for agents to discover the surrounding context, e.g., by reading tuples describing the most recent sensed values. Reactive applications requiring on-the-fly reprogramming of sensor nodes (e.g., fire monitoring) are thus an ideal target scenario for Agilla.

ATaG. The Abstract Task Graph (ATaG) [25] is a programming framework providing a mixed declarative-imperative approach. The notions of *abstract task* and *abstract data item* are at the core of the ATaG programming model. A task is a logical entity encapsulating the processing of one or more data items, which represent the information. The flow of information between tasks is defined in terms of *abstract channels* used to connect each data item to the tasks that produce or consume it. To exchange data among tasks, programmers are provided with the abstraction of a shared data pool where tasks can output data or be asynchronously notified when some data of interest is available. This style of interaction is similar to tuple spaces for mobile networks when using reactive operations, as described in Section 1.2.2, although in ATaG this is limited to triggering notifications when other processes perform a write, and the data of interest is determined solely based on its type, rather than an arbitrary pattern. The ability to isolate different processing steps in different tasks makes ATaG suited to control applications requiring multi-stage processing, e.g., road traffic control [26].

FACTS. Terfloth et al. [27] propose a middleware platform inspired by logical reasoning in expert systems. In FACTS, modular pieces of processing instructions called *rules* describe how to handle information. These are specified using a dedicated language called *RDL* (ruleset definition language), whereas data is specified in a special format called a *fact*. The appearance of new facts trigger the executions of one or more rules, that may generate new facts or remove existing ones. Facts can be shared among different nodes. The basic communication primitives in FACTS provide one-hop data sharing. Facts are reminiscent of tuples as a way of structuring the application data, while the triggering of rules in response to new facts is similar to the execution of reactions in mobile tuple space middleware. Reactive applications such as fence monitoring [28] are easily implemented using FACTS, thanks to the condition-action rules programmers can specify.

Hood. The programming primitives provided by Hood [29] revolve around the notion of neighborhood. Constructs are provided to identify a subset of a node's physical neighbors based on application criteria, and to share data with them. A node exports information in the form of *attributes*. Membership in a programmer-specified neighborhood is determined using *filters*. These are boolean functions that examine a node's attributes and determine, based on their values, whether the remote node is to become part of the considered subset. If so, a mirror for that particular neighbor is created on the local node. The mirror contains both *reflections*, i.e., lo-

	Abstract Regions [23]	**Agilla** [24]	**ATaG** [25]	**FACTS** [27]	**Hood** [29]	**TeenyLIME** [30]
Reactive vs. Proactive	Proactive	Both	Reactive	Reactive	Proactive	Both
Push vs. Pull	Push	Both	Push	Push	Push	Both
Data Filtering	Shared variables	Pattern matching	Named channels	Fact constraints	Application-level filtering	Pattern matching w/ value constraints
Data Processing	Reduce operator	N/A	N/A	N/A	N/A	N/A
Scope	Programmer-defined	Local	Programmer-defined	One-hop	One-hop	One-hop

Table 1.2 Features of tuple space systems for wireless sensor networks.

cal copies of the neighbor's attributes that can be used to access the shared data, and *scribbles*, which are local annotations about a neighbor. Hood can be seen as a tuple space system where **out** operations are used to replicate local information on neighboring nodes, and filters take care of the matching functionality. Thanks to the support of multiple independent neighborhoods, Hood is applicable in diverse settings ranging from target tracking applications to localization mechanisms and MAC protocols [29].

TeenyLIME. Inspired by LIME, TeenyLIME [30] offers a Linda-like interface to programmers. To make the tuple space paradigm blend better with the asynchronous programming model of most WSN operating systems (e.g., TinyOS [31]), operations are non-blocking and return their results through a callback. In TeenyLIME, tuples are shared among neighboring nodes. *Reactions* are provided to allow for asynchronous notifications in case some specific piece of data appears in the shared tuple space. In addition, several WSN-specific features are made available to better address the requirements of sensor network applications. For instance, a notion of *capability tuple* is provided to enable on-demand sensing. This can save the energy required to keep sensed information up to date in the shared tuple space in the absence of any data consumer. Similarly to Hood, TeenyLIME reaches into the entire stack, providing constructs to develop full-fledged applications as well as system-level mechanisms, e.g., routing protocols. However, TeenyLIME specifically targets sense-and-react applications, (e.g., HVAC in buildings [32]), where its reactive and WSN-specific features provide a significant asset.

1.3.2 Discussion

The approaches outlined above are possibly more heterogeneous than those we discussed in Section 1.2. Table 1.2 summarizes the most important similarities and differences.

Reactive vs. Proactive Operations. The ability to react to external stimuli is of paramount importance in WSNs. Likewise, being able to proactively influence the data shared with other processes is fundamental to achieve coordination through the tuple space. Agilla and TeenyLIME provide both modes of operations, i.e., the tra-

ditional tuple space operations to express proactive interactions along with a notion of *reaction* inspired by similar functionality in the mobile setting, as described in Section 1.2.2. The semantics provided, however, are generally weaker. For instance, no guarantees are provided on whether a reaction can fire multiple times for the same tuple. Similarly, the RDL language in FACTS allows for the specification of programmer-provided conditions for a rule to fire. Nevertheless, in these cases, implementing reactive operations raises issues with their semantics that are difficult to face on resource constrained devices. For instance, in the presence of multiple reactions (rules) being triggered simultaneously, the aforementioned systems provide no guarantees w.r.t. the order they are scheduled. This issue is partially solved in ATaG by forcing channels to behave in a FIFO manner, and imposing a round-robin schedule across different channels. Differently, both Abstract Regions and Hood provide only proactive operations. In Abstract Regions, the only way of observing a change in the shared data is to proactively read the value of a variable. Similarly, in Hood the local reflections must be manually inspected to recognize some change in the shared data. Both systems are therefore significantly less expressive, and lead to more cumbersome programming whenever some form of reactive behavior is required.

Proactive Push vs. Proactive Pull. Although the approaches described above do provide some notion of data sharing, none of them completely abstracts away the location of data. As a result, proactive operations can occur with different modes of operations. Existing systems operate either in a push manner—where the data producers move the data towards the data consumers—or in a pull manner, i.e., where data stays with the data producer until a consumer explicitly retrieves them. ATaG and FACTS provide only push primitives. In the latter, for instance, facts can be written at remote nodes but cannot be retrieved from them. The rest of the systems described, instead, provide both modes of operation. Agilla and TeenyLIME provide simple variations of the traditional tuple space API, offering pull operations such as **rd** as well as push ones, e.g., **out**. In Abstract Regions, remote variables can be read and written, which provides a way to describe both push and pull interactions. Likewise, although the normal operational mode in Hood is to periodically push a node's attributes towards its neighbors using a form of beaconing, a node can also request an on-demand update of a neighbor's data, a functionality analogous to a pull operation.

Data Filtering. The pattern matching mechanism in traditional tuple spaces provide an expressive way to identify the data of interest. Agilla and TeenyLIME retain the same by-field, type-based matching semantics, but also provide the ability to specify further conditions on the value of the data themselves other than simple equality. For instance, in TeenyLIME it is straightforward to specify a pattern identifying a tuple whose first field is a 16 bit integer with a value above a given threshold. A similar feature is also present in FACTS, though the processing to enforce the condition on the fact repository is partially left to the application programmer. The other systems considered, instead, provide simpler—but less expressive—mechanisms for identifying the data of interest. In Abstract Regions, variable names are, in a sense, the

only data filtering mechanism available; similar considerations hold for ATaG and Hood. Furthermore, in all these latter approaches the type of data to be shared is essentially decided at compile time. Therefore, the conditions to filter data cannot be changed by the application while the system is running.

Data Processing. Embodying constructs dedicated to elaborate data in the programming model can help WSN programmers to deal with commonly seen processing patterns, e.g., averaging the reading of a number of sensor nodes. The only work explicitly addressing this concern is Abstract Regions. The presence of the `reduce()` operator in the API allows programmers to apply a given operator to all shared variables of a given type, and possibly to assign the result to a further shared variable. The set of operators available, however, strictly depends on the specific implementation used in support of the region at hand. Some operators, (e.g., `MAX`) can be straightforwardly implemented for various kinds of regions. Instead, others need customized, per-region implementations: for instance, computing the correct outcome of `AVG` requires keeping track of duplicates.

Scope. In Section 1.2.2, it was observed that providing a shared tuple space spanning all the nodes in the system is often prohibitive in mobile networks. This issue is brought to an extreme in WSNs, due to the limited communication abilities of typical WSN devices. Consequently, in the current state of the art, the span of the shared memory space turns out to be quite limited. Most approaches provide one-hop data sharing, or even just local sharing as in the case of Agilla. Notable exceptions are Abstract Regions and ATaG. In both cases, however, dedicated routing support must be provided to enable data sharing beyond the physical neighborhood of a node. In Abstract Regions, each different region requires a dedicated implementation. For instance, an n-hop region can be implemented using limited flooding, whereas a region defined in terms of geographic boundaries can be implemented using GPSR [33]. This same protocol is used in ATaG when the span of the data pool is defined based on the locations of nodes assigned a given task. Alternatively, the data pool in ATaG can be shaped according to logical attributes of the nodes (e.g., their type) using the routing provided by Logical Neighborhoods [26].

1.3.3 Tuple Spaces in Action

This section illustrates how the driving concepts of tuples spaces can be applied to a paradigmatic sensor network scenario. We discuss the implementation of a sense-and-react application using TeenyLIME. In similar scenarios, nodes hosting actuators are deployed alongside sensing devices. The system is designed to react to stimuli gathered by sensors and affect the environment by means of actuators.

Scenario and Requirements. Consider an application for *emergency control in buildings* whose main functionality is to provide guidance and first response under exceptional circumstances, e.g., in case of fire. The application logic features four main components, illustrated in Figure 1.3. The *user preferences* represent the

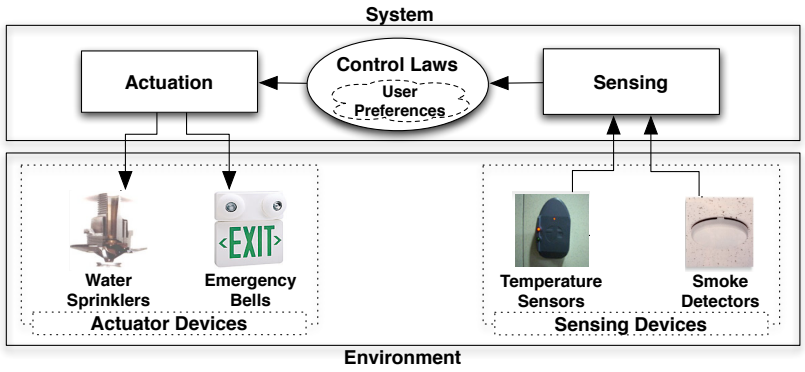

Fig. 1.3 Emergency control in buildings.

high-level system goals, in this case, the need to limit fire spreading. *Sensing devices* gather data from the environment and monitor relevant variables. In this case, smoke and temperature detectors recognize the presence of a fire. *Actuator devices* perform actions affecting the environment under control. In the scenario at hand, water sprinklers and emergency bells are triggered in case of fire. *Control laws* map the data sensed to the actions performed, to meet the user preferences. In this case, a (simplified) control loop may activate emergency bells when the temperature increases above a safety threshold, but operate water sprinklers only if smoke detectors actually report the presence of fire.

The characteristics of the scenario make programming similar systems a challenging task:

- *Localized computations* [34] must be privileged to keep processing close to where sensing or actuation occurs. It is indeed unreasonable to funnel all the sensed data to a single base-station, as this may negatively affect latency and reliability without any significant advantage [35].
- *Reactive interactions*, i.e., actions that automatically fire based on external conditions, play a pivotal role. In this case, a temperature reading above a safety threshold must trigger an action on the environment.
- *Proactive interactions*, however, are still needed to gather information and fine tune the actuation about to occur. For instance, the sprinklers in the building ask for smoke readings before taking any action.

The stateful nature of the tuple space abstraction naturally lends itself to addressing the above requirements, as the actions taken on the environment must be decided based on the current sensed state. In addition, tuple spaces make it easier to express the coordination required in these scenarios, e.g., since they provide both proactive and reactive operation. Alternative paradigms, such as publish-subscribe, only partially meet the above challenges, being essentially biased towards either reactive or proactive interactions.

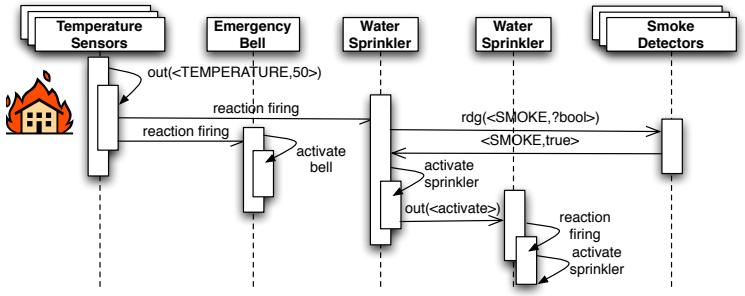

Fig. 1.4 Sequence of operations to handle a fire. Once notified about increased temperature, a node controlling water sprinklers queries the smoke detectors to verify the presence of fire. If necessary, it sends a command activating nearby sprinklers.

Design and Implementation. In our design, sensed data and actuating commands take the form of tuples. These are shared across nodes to enable coordination of activities and data communication. Figure 1.4 illustrates how proactive and reactive interactions in TeenyLIME are used to deal with the possibility of a fire. Both emergency bells and water sprinklers have a reaction registered on their neighbors, watching for temperature tuples over a given threshold. This is accomplished using TeenyLIME's value matching functionality, described in Section 1.3.2. Temperature sensors periodically take a sample and pack it in a tuple, which is then stored in the local tuple space. This operation, by virtue of TeenyLIME's one-hop sharing, automatically triggers all the aforementioned reactions in case of a positive match.

However, different types of actuator nodes behave differently when high temperatures are detected. The node hosting the emergency bell immediately activates its attached device. Instead, the water sprinkler node proceeds to verify the presence of fire, as shown in Figure 1.4. The latter behavior, specified as part of the reaction code, consists of proactively gathering the readings from nearby smoke detectors, using a **rdg** over the entire shared tuple space. If fire is reported, the water sprinkler node requests activation of nearby sprinklers through a two-step process that relies on reactions as well. The node requesting actuation inserts a tuple representing the command on the nodes where the activation must occur. The presence of this tuple triggers a locally-installed reaction delivering the activation tuple to the application, which reads the tuple fields and operates the actuator device accordingly.

Based on the above processing, smoke detectors need not be monitored continuously: their data is accessed only when actuation is about to occur. However, when a sensed value is requested (e.g., by issuing a **rd**) fresh-enough data must be present in the tuple space. If these data are only seldom utilized, the energy required to keep tuples fresh is mostly wasted. An alternative is to require that the programmer encodes requests to perform sensing on-demand and return the result. To avoid this extra programming effort, TeenyLIME's capability tuples can be used as described in Section 1.3.1. Here, a capability tuple is used as a placeholder to represent a node's ability to produce data of a given type, without keeping the actual data in

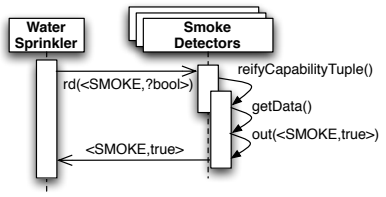

Fig. 1.5 Processing of capability tuples.

the tuple space. When a query is remotely issued with a pattern matching a capability tuple, a dedicated event is signaled to the application. In response to this, the application takes a fresh reading and outputs the actual data to the tuple space. The sequence of operations is depicted in Figure 1.5. Note how, from the perspective of the data consumer, nothing changes. Instead, at the data producer, capability tuples enable considerable energy savings as the readings are taken only on-demand.

The simple yet expressive programming primitives provided by TeenyLIME allow programmers to express complex interactions in a few lines of code. [30] compares TeenyLIME-based implementations of various applications and system-level mechanisms, e.g., routing protocols, against functionally equivalent nesC implementations. Results indicate that TeenyLIME yields cleaner and simpler implementations. For instance, the number of lines of code written by programmers using TeenyLIME is usually half of the corresponding counterpart implemented in nesC.

1.4 Summary and Outlook

Tuple spaces were originally invented as a computational and programming model for parallel computing. Later on, companies such as IBM proposed tuple spaces as a programming model for distributed computing. The next natural step is, therefore, for the programming models inspired by tuple spaces to follow the evolution of networking towards untethered, wireless communication. In this context, simplicity of the programming interface and decoupling in time and space are the most significant advantages. Nevertheless, the characteristics of wireless communication demand a re-thinking and extension of the base model made popular by Linda.

This chapter concisely presented the state-of-the-art concerning middleware platforms based on the tuple space abstraction and expressly designed for wireless scenarios. By analyzing and comparing representative systems it provided the reader with a wide perspective on the efforts in this specific field. Moreover, by exemplifying the use of tuple spaces in realistic application case studies, it conveyed concretely the power of this abstraction.

Although the presentation is structured along the two application scenarios defined by mobile computing and wireless sensor networks, this does not necessarily imply that these two realms must be treated separately. Indeed, in our own work we

explored a two-tier approach where sensor data is available to mobile data sinks, from which it is available to other sinks in range. In this model and system, called TinyLIME [36], the transiently shared tuple space originally introduced by LIME is the unifying abstraction enabling seamless access to information spanning different kinds of networks. Indeed, bridging the physical world where users live and move with the virtual world enabled by sensing and actuating wireless devices is a new frontier of computing [37], and one where the simple and effective data sharing paradigm fostered by tuple spaces can give a fundamental contribution.

References

1. Malone, T., Crowston, K.: The Interdisciplinary Study of Coordination. ACM Computing Surveys **26**(1) (March 1994) 87–119
2. Busi, N., Zavattaro, G.: Publish/subscribe vs. shared dataspace coordination infrastructures: Is it just a matter of taste? In: Proc. of the 10^{th} Int. Wkshp. on Enabling Technologies (WETICE). (2001)
3. Ceriotti, M., Murphy, A.L., Picco, G.P.: Data Sharing vs. Message Passing: Synergy or Incompatibility? An Implementation-driven Case Study. In: Proc. of the 23^{rd} Symposium on Applied computing (SAC). (2008)
4. Li, G., Cheung, A., Hou, S., Hu, S., Muthusamy, V., Sherafat, R., Wun, A., Jacobsen, H.A., Manovski, S.: Historic data access in publish/subscribe. In: Proc. of the 1^{st} Int. Conf. on Distributed Event-based Systems (DEBS). (2007)
5. Gelernter, D.: Generative communication in Linda. ACM Computing Surveys **7**(1) (1985)
6. Rowstron, A.: WCL: A coordination language for geographically distributed agents. World Wide Web Journal **1**(3) (1998) 167–179
7. Roman, G.C., Murphy, A.L., Picco, G.P.: Coordination and Mobility. In Omicini, A., Zambonelli, F., Klusch, M., Tolksdorf, R., eds.: Coordination of Internet Agents: Models, Technologies, and Applications. Springer (2000) 254–273 Invited contribution.
8. Picco, G., Murphy, A., Roman, G.C.: LIME: Linda Meets Mobility. In Garlan, D., ed.: Proc. of the 21^{st} International Conference on Software Engineering (ICSE'99), Los Angeles, CA, USA, ACM Press (May 1999) 368–377
9. Murphy, A.L., Picco, G.P., Roman, G.C.: LIME: A coordination model and middleware supporting mobility of hosts and agents. ACM Trans. on Software Engineering and Methodology (TOSEM) **15**(3) (2006)
10. Costa, P., Mottola, L., Murphy, A.L., Picco, G.P.: TeenyLIME: Transiently shared tuple space middleware for wireless sensor networks. In: Proc. of the 1^{st} Int. Wkshp. on Middleware for Sensor Networks (MidSens). (2006)
11. `www.almaden.ibm.com/cs/TSpaces`
12. Davies, N., Friday, A., Wade, S.P., Blair, G.S.: L^2imbo: A distributed systems platform for mobile computing. Mob. Netw. Appl. **3**(2) (1998) 143–156
13. Fuggetta, A., Picco, G.P., Vigna, G.: Understanding code mobility. IEEE Trans. Softw. Eng. **24**(5) (1998)
14. Handorean, R., Roman, G.C.: Service provision in ad hoc networks. In: Proceedings of the 5^{th} International Conference on Coordination Models and Languages (COORDINATION). LNCS 2315, Springer (2002)
15. Murphy, A.L., Picco, G.P.: Using LIME to Support Replication for Availability in Mobile Ad Hoc Networks. In: Proceedings of the 8^{th} International Conference on Coordination Models and Languages (COORD06), Bologna (Italy) (June 2006)
16. Picco, G.P., Buschini, M.L.: Exploiting Transiently Shared Tuple Spaces for Location Transparent Code Mobility. In: Proceedings of the 5^{th} International Conference on Coordination Models and Languages (COORDINATION). LNCS 2315, Springer (2002)

17. Julien, C., Roman, G.C.: EgoSpaces: Facilitating Rapid Development of Context-Aware Mobile Applications. IEEE Trans. Softw. Eng. **32**(5) (2006)
18. Mamei, M., Zambonelli, F.: Programming Pervasive and Mobile Computing Applications with the TOTA Middleware. In: Proceedings of the 2^{nd} IEEE International Conference on Pervasive Computing and Communications (PerCom). (2004)
19. Murphy, A., Picco, G.: Using Coordination Middleware for Location-Aware Computing: A LIME Case Study. In: Proc. of the 6^{th} Int. Conf. on Coordination Models and Languages (COORD04). LNCS 2949, Springer (February 2004) 263–278
20. Balzarotti, D., Costa, P., Picco, G.P.: The LighTS Tuple Space Framework and its Customization for Context-Aware Applications. International Journal on Web Intelligence and Agent Systems (WIAS) **5**(2) (2007) 215–231
21. Floyd, S., Jacobson, V., McCanne, S., Liu, C.G., Zhang, L., Zhang, L.: A reliable multicast framework for light-weight sessions and application level framing. SIGCOMM Comput. Commun. Rev. **25**(4) (1995)
22. Gummadi, R., Gnawali, O., Govindan, R.: Macro-programming wireless sensor networks using Kairos. In: Proc. of the 1^{st} Int. Conf. on Distributed Computing in Sensor Systems (DCOSS). (2005)
23. Welsh, M., Mainland, G.: Programming sensor networks using abstract regions. In: Proc. of 1^{st} Symp. on Networked Systems Design and Implementation (NSDI). (2004)
24. Fok, C.L., Roman, G.C., Lu, C.: Rapid development and flexible deployment of adaptive wireless sensor network applications. In: Proc. of the 25^{th} Int. Conf. on Distributed Computing Systems (ICDCS). (2005)
25. Bakshi, A., Prasanna, V.K., Reich, J., Larner, D.: The Abstract Task Graph: A methodology for architecture-independent programming of networked sensor systems. In: Workshop on End-to-end Sense-and-respond Systems (EESR). (2005)
26. Mottola, L., Pathak, A., Bakshi, A., Picco, G.P., Prasanna, V.K.: Enabling scope-based interactions in sensor network macroprogramming. In: Proc. of the the 4^{th} Int. Conf. on Mobile Ad-Hoc and Sensor Systems (MASS). (2007)
27. Terfloth, K., Wittenburg, G., Schiller, J.: Facts - a rule-based middleware architecture for wireless sensor networks. In: Proc. of the 1^{st} Int. Conf. on Communication System Software and Middleware (COMSWARE). (2006)
28. Wittenburg, G., Terfloth, K., Villafuerte, F.L., Naumowicz, T., Ritter, H., Schiller, J.: Fence monitoring - experimental evaluation of a use case for wireless sensor networks. In: Proc. of the 4^{th} European Conf. on Wireless Sensor Networks (EWSN). (2007)
29. Whitehouse, K., Sharp, C., Brewer, E., Culler, D.: Hood: a neighborhood abstraction for sensor networks. In: Proc. of 2^{nd} Int. Conf. on Mobile Systems, Applications, and Services (MOBISYS). (2004)
30. Costa, P., Mottola, L., Murphy, A.L., Picco, G.P.: Programming wireless sensor networks with the TeenyLIME middleware. In: Proc. of the 8^{th} ACM/USENIX Int. Middleware Conf. (2007)
31. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., Pister, K.: System architecture directions for networked sensors. In: ASPLOS-IX: Proc. of the 9^{th} Int. Conf. on Architectural Support for Programming Languages and Operating Systems. (2000)
32. Deshpande, A., Guestrin, C., Madden, S.: Resource-aware wireless sensor-actuator networks. IEEE Data Engineering **28**(1) (2005)
33. Karp, B., Kung, H.T.: Greedy perimeter stateless routing for wireless networks. In: Proc. of the 6^{th} Int. Conf. on Mobile Computing and Networking (MobiCom). (2000)
34. Estrin, D., Govindan, R., Heidemann, J., Kumar, S.: Next century challenges: scalable coordination in sensor networks. In: Proc. of the 5^{th} Int. Conf. on Mobile computing and networking (MOBICOM). (1999)
35. Akyildiz, I.F., Kasimoglu, I.H.: Wireless sensor and actor networks: Research challenges. Ad Hoc Networks Journal **2**(4) (2004)
36. Curino, C., Giani, M., Giorgetta, M., Giusti, A., Murphy, A.L., Picco, G.P.: Mobile data collection in sensor networks: The TinyLime middleware. Elsevier Pervasive and Mobile Computing Journal **4**(1) (2005)
37. ITU: The Internet of Things. Technical report, International Telecommunication Union (2005)