Pervasive Games in a Mote-Enabled Virtual World Using Tuple Space Middleware

Luca Mottola
Dipartimento di Elettronica ed Informazione
Politecnico di Milano, Italy
mottola@elet.polimi.it

Amy L. Murphy
Department of Informatics
University of Lugano, Switzerland
amy.murphy@unisi.ch

Gian Pietro Picco
Dipartimento di Elettronica ed Informazione
Politecnico di Milano, Italy
picco@elet.polimi.it

ABSTRACT

Pervasive games are a new and exciting field where the user experience benefits from the blending of real and virtual elements. Players are no longer confined to computer screens. Rather, interactions with devices embedded within the real world and physical movements become an integral part of the gaming experience. Several prototypes of pervasive games have been proposed by both industry and academia. However, in such games the issues arising from the integration of players and real world, the management of the context surrounding the players, and the need for communication and distributed coordination are often addressed in an ad-hoc fashion. Therefore, the underlying software fabric is often not reusable, ultimately slowing down the diffusion of pervasive games.

In this paper we describe the design and implementation of a pervasive game on top of TinyLIME, a middleware system supporting data sharing among mobile and embedded devices. By illustrating the design of a pervasive game we developed, we argue concretely that the programming abstractions supported by TinyLIME greatly simplify the data and context management characteristics of pervasive games, and provide an effective and reusable building block for their development.

TinyLIME was originally designed to support applications where mobile users collect data from sensors scattered in the physical environment. We build upon this capability to put forth a second contribution, namely, the use of wireless sensor devices (or *motes*) as a computing platform for pervasive games. Besides reporting physical data for the sake of the game, we use motes to store information relevant to the game plot, e.g., *virtual* objects. Motes are typically very small in size, and therefore can be hidden in the environment, enhancing the sense of immersion in a virtual world. To the best of our knowledge, this original use of wireless sensor devices is novel in the scientific and gaming literature. Furthermore, it is naturally supported by TinyLIME, yielding a unified programming abstraction that spans the heterogeneous gaming platform we propose.

1. INTRODUCTION

Before the advent of the computer era, games were based on interactions among humans or between humans and the physical world. However, thanks to impressive technological advances, computer games have now become a cornerstone of the entertainment industry. In computer games, the players are usually confined to the use of keyboards, mice, and joysticks to interact with a completely virtual environment. Conversely, a more exciting game experience can be provided by bringing back physical movements and social interactions in computer games [15]. This has given rise to a new gaming genre, called *pervasive games*, where players are no longer limited to a purely virtual environment. Rather, it is the interaction with the real world or devices embedded within it that poses new challenges to the players, enhancing their gaming experience.

Several prototypes of pervasive games (e.g., [3, 8, 26, 27]) have been developed, each focusing on different kinds of interactions between the players and the surrounding physical environment. Therefore, much attention has been paid to the technological aspects regarding hardware platforms, usability, and human-computer interaction. Conversely, less effort has been spent on the software infrastructure enabling the distributed processing needed by pervasive games. Several challenging issues arise when analyzing the requirements of pervasive games:

- Pervasive games need a software infrastructure able to provide seamless *integration* between the players' devices and the devices embedded within the real world.
- The system must be *context-aware* [28], as the players actions and decisions, as well as the game evolution, are affected by the physical context.
- Distributed *coordination* and *communication* is needed at two levels: i) *among players*, to enable their collaboration towards a common goal, and ii) *to enforce the intended game behavior* and support the evolution of the game "behind the scenes", i.e., without the players being explicitly aware of what is happening in the game.

In this work we show how the above challenges can be tackled effectively by using TinyLIME [4, 5], a middleware originally developed for pervasive computing scenarios involving mobile users and wireless sensor networks [2]. Our claim is that TinyLIME provides programming abstractions that naturally support the kind of context-aware interaction required by pervasive games, and effectively hides the complexity of dealing with heterogeneous devices.

We support concretely this claim by reporting the design and implementation of a simple game called "Save the Princess!". A distinctive trait of our game is an original use of wireless sensor network devices. These tiny devices, often called *motes*, normally enable measurement of physical parameters such as temperature or light, and can be programmed to perform simple local computational tasks and manage the communication necessary to report

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

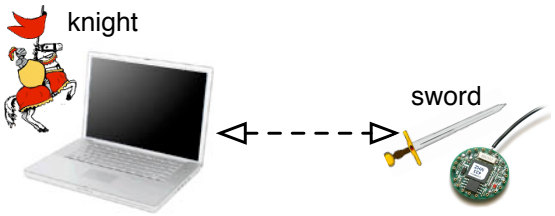


Figure 1: Interaction between a user and a virtual object, represented by a mote.

data. In the context of our game, mote programmability is exploited to support the creation of a virtual environment, by programming motes to store the state of *virtual* objects, characters, and locations. As shown in Figure 1, the application of a mobile user close to a mote communicates with it, and displays the corresponding information about the virtual objects to the user, therefore enabling context-aware virtual interactions. Because of their small size, motes can be hidden in the environment allowing them to "disappear" [24], therefore enhancing the illusion of a virtual world.

The game requires mobile users to cooperate to achieve a common goal, and game dynamics are determined by both player-to-player interactions as well as interaction with the physical and virtual environment, as shown in Figure 2. Players communicate *directly* among themselves in an opportunistic fashion, based on the availability of connectivity. Moreover, they can communicate *indirectly* by storing information in the virtual world around them. For instance, a player can give a virtual object she is holding either directly to another player, or leave it behind (i.e., stored transparently on a nearby mote) for another player to pick up later. The game and its requirements, along with the details of the hardware and software platform, are illustrated in Section 2.

The TinyLIME middleware, described in Section 3, supports pervasive game development through the abstraction of a *tuple space* [10]. Applications communicate by writing tuples (elementary data items) in the tuple space, and by reading them by specifying the pattern of data they are interested in. Therefore, the tuple space can be regarded as a sort of data repository. To deal with mobility and context changes, TinyLIME provides the programmer with the illusion of a single, shared tuple space containing the data contributed by all the devices in range. The middleware takes care of maintaining this abstraction consistent with respect to the dynamics of the system. This means that the programmer does not need to be explicitly aware of the current network configuration,

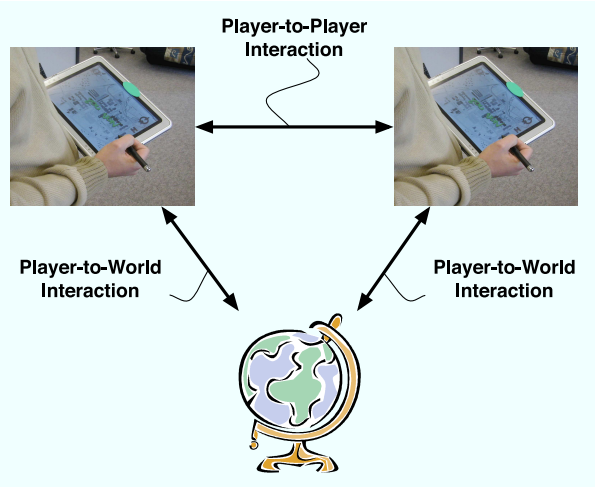


Figure 2: Direct and indirect interactions among players.

rather it is the middleware that transparently deals with the mechanics of data access based on the current connectivity. Moreover, TinyLIME extends the original tuple space model with the notion of *reactions*. In contrast with reading operations, reactions enable applications to declare their interest in the presence of a tuple with a specified pattern. When a matching tuple is actually found in the tuple space, a callback is asynchronously executed. This ability to react to new data is of paramount importance in the dynamic environment we target, and supports the reactive behavior necessary to model context-aware interactions. These and other capabilities of the TinyLIME middleware are demonstrated "in action" in Section 4, where we illustrate the design and implementation of our pervasive game, and show that TinyLIME indeed provides natural abstractions simplifying game development.

Finally, since TinyLIME was originally designed as a data collection middleware for wireless sensor networks, its focus was on retrieving data from sensors, essentially regarded by the user as "read-only" devices. Instead, to support pervasive games the application must be able to write data to the sensor devices, e.g., to change the state of a virtual object. This required changes to the original TinyLIME implementation, whose rationale and impact are concisely described in Section 5.

In short, this paper puts forth two main contributions:

1. We propose an original use of wireless sensor devices as tiny computing platforms that can be hidden in the environment and can *actively* contribute to enrich the gaming experience by *hosting game-related data*. To the best of our knowledge, this use of motes is unprecedented in the gaming and scientific literature.

2. We show how the TinyLIME middleware, originally designed to support mobile data collection from sensors, naturally supports the *context-aware interactions* necessary for pervasive game development and effectively masks the hardware and software heterogeneity.

These contributions are concretely supported and exemplified by our proof-of-concept pervasive game, described next.

2. SAVE THE PRINCESS!

The Game. "Save the Princess!" tells the age-old story of a princess imprisoned in a castle. To save her, the players must beat the black knight, who kidnapped the princess. To this end, they must collect a set of *ingredients* to prepare a magical poison. Once the poison is ready, all the players together must face the black knight in the final battle.

Players have different *abilities* and are divided in teams according to the *role* they incarnate: possible roles are *wizards*, *dwarfs*, and *knights*. In addition, each team has a *leader* whose abilities are extended beyond those germane to her game role. Wizards are the only players able to collect ingredients. These must then be brought to a cauldron, whose location is a-priori unknown. In addition, ingredients might not be immediately available for pickup: an *ogre* or a *skeleton* could hide the ingredients or occupy the passage to the location where the ingredient is located. Ogres can be beaten only by dwarfs, whereas skeletons can be defeated only by knights. Whether a knight or a dwarf actually wins a battle depends on a player's current abilities (and on a bit of randomness).

Beyond the ingredients needed for the poison, players can pick up any other *object* they find. These objects can be *exchanged* among players, immediately *used* to augment the abilities of a player, or held in the hope of *combining* them with other objects found

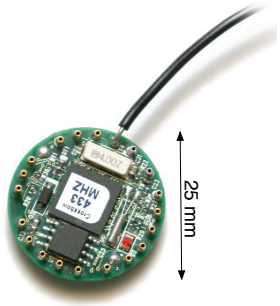


Figure 3: A coin-size Crossbow MICA2DOT mote.

later to form more powerful objects. For instance, a knight could find a sword in a certain location, and using that sword he can attack the skeletons. Although objects can be used multiple times, once used for the first time they can no longer be combined with others. The player can therefore decide not to use the sword immediately, and later combine it with a shield. In this case, the player will defeat the skeletons much more easily, but she must survive until the shield is found.

Hardware and Software Platform. Players are equipped with laptop computers with an IEEE 802.11 wireless interface, forming a mobile ad-hoc network [20]. Clearly, this is only for the sake of prototyping, and more sophisticated devices (e.g., head-mounted displays) can be used. The game, as well as the TinyLIME middleware on top of which the game is developed, are written in Java.

The real world is augmented with computing facilities using the Crossbow MICA2DOT motes [1], shown in Figure 3. These are tiny devices (about the size of a coin) equipped with a small amount of volatile memory, a short range radio interface and an 8-bit microprocessor. Motes are fairly unobtrusive, and can be deployed so that players are not aware of their exact location. In addition, they feature a wide range of real sensors, from acoustic to vibration sensors and GPS receivers, and can also control simple external devices such as buzzers or LEDs. These features can be used to enable additional interactions between the players and the physical world, using real-world data and actuators to further improve the gaming experience. The motes run the TinyOS [13] operating system, based on the nesC [9] language. It is an event-driven programming language targeted to resource constrained devices.

Communication among player laptops is naturally achieved using the Wi-Fi interfaces. Conversely, communication between laptops and motes is achieved by using an additional mote attached to each laptop via a serial line. This mote does not perform any relevant operations, it simply acts as an antenna between the laptop and the motes deployed in the environment.

Requirements. The pervasive game challenges we pointed out in the introduction are all present in "Save the Princess!". First, we need to face the integration between the player laptops and the motes. The former are powerful devices equipped with rich user interfaces, easily rechargeable, and able to run a wide range of execution environments. Conversely, the motes do not feature any user interface, making application debugging difficult, and run a severely constrained operating system with no support for dynamic memory allocation or high-level programming constructs such as multithreading. The only communication facility provided out-of-the-box is a simple form of message passing to other motes within the wireless communication range. In this setting there is a clear need for unifying programming abstractions able to hide the underlying heterogeneity of the system.

Pervasive games are inherently state-based applications. Moreover, this state is distributed, partly located on the players' devices and partly on the mote devices. The application state, along with connectivity relationships, determine the context around the players. Game developers need to deal with and reason about the distributed application and context state, and react when changes are observed. For instance, consider a mote holding data representing a magical ingredient. When a wizard is in proximity of this mote, he must be informed of the possibility to pick up the ingredient. Without appropriate programming abstractions, a programmer must explicitly poll for the presence of nearby motes and explicitly query them, therefore wasting programming effort as well as communication bandwidth. The management of application data and context is also key to achieve coordination and communication among players both directly and through the (virtual) environment.

The transparent sharing of data and context information provided by TinyLIME, coupled with its asynchronous reactive features, greatly simplify the programming task. Before arguing concretely in support of this statement by illustrating our design, we provide a concise introduction to the TinyLIME model and middleware.

3. THE TINYLIME MIDDLEWARE

TinyLIME is a data sharing middleware based on LIME [18], which in turn adapted the Linda tuple space model [10] to mobile environments. In this section we outline the characteristics of TinyLIME that are most important for the development of pervasive games. A more comprehensive overview of TinyLIME can be found in [4].

Tuple Spaces. A *tuple* is an elementary data structure composed of an ordered sequence of typed fields, whereas a *tuple space* is a multiset of tuples. A sample tuple is:

$$\langle \text{“Milan”}, 80, 9.75 \rangle \tag{1}$$

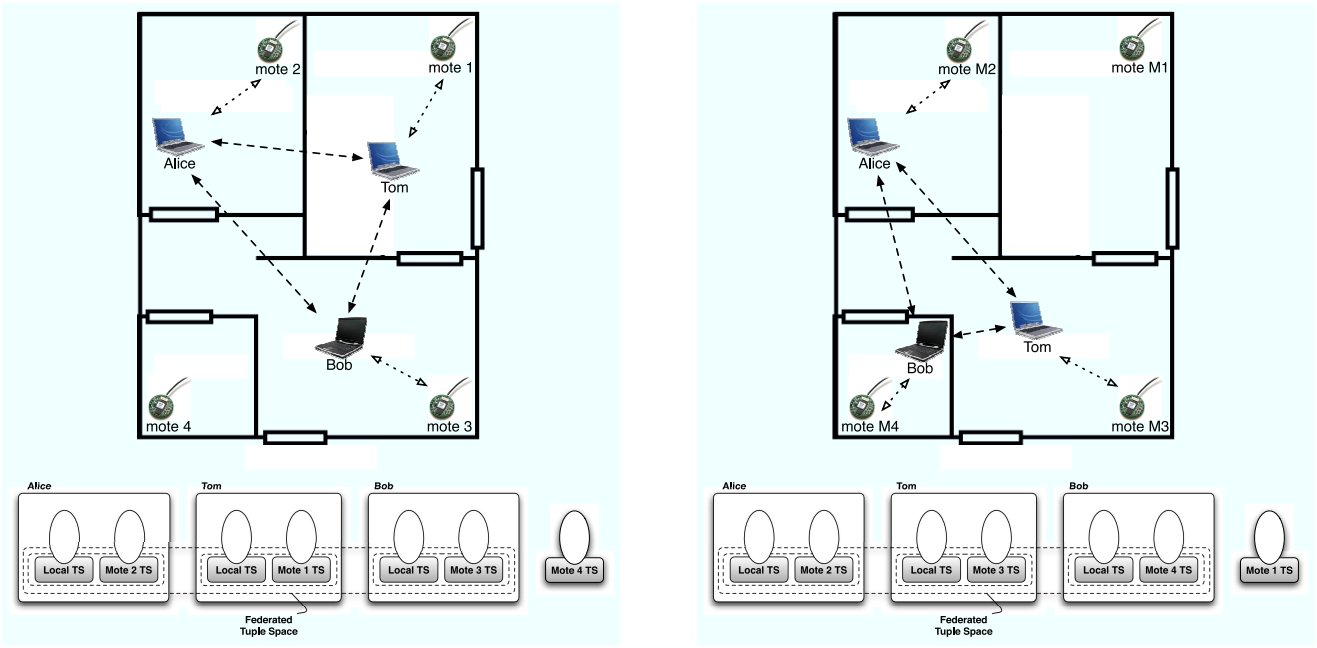
Processes interact by reading or writing tuples from/to a tuple space logically shared by a set of processes. Read operations can be performed in this tuple space using the $\mathbf{rd}(p)$ operation, which takes a pattern p as a parameter specifying the type of data desired. A sample pattern is:

$$\langle \text{“Milan”}, ?integer, ?float \rangle \tag{2}$$

The pattern itself is a tuple whose fields can be either *actual* or *formal*: actuals constrain the values that must be present in the corresponding fields of the tuples returned by the operation. The first field in (2) is an example of an actual. Instead, formals act like "wild cards" and are matched against actual values when selecting a tuple from the tuple space. The last two fields of (2) are formals. As such, the pattern in (2) matches the tuple in (1). Additional operations insert a tuple in the tuple space by means of the $\mathbf{out}(t)$ operation, and remove tuples from the tuple space using the $\mathbf{in}(p)$ operation.

The tuple space model inherited from LIME also allows the registration of *reactions* on the shared tuple space. These are code fragments to be executed when a tuple matching a specific pattern is found anywhere in the tuple space, thus providing the developer with the ability to monitor changes in the underlying tuple space. The code executed when a reaction fires can perform an arbitrary sequence of actions, including further operations on the shared tuple space.

Mobility, Context, Sensors. TinyLIME inherits LIME's view of application components as software agents installed on mobile hosts. In LIME, each agent has a local tuple space permanently attached.



(a) Initial configuration.

(b) New configuration after Bob and Tom moved.

Figure 5: The TinyLIME federated tuple space changes according to connectivity: motes in range of a mobile device are seen as software agents residing on that device. If no mobile device is in range of a mote, it is not part of the federated tuple space.

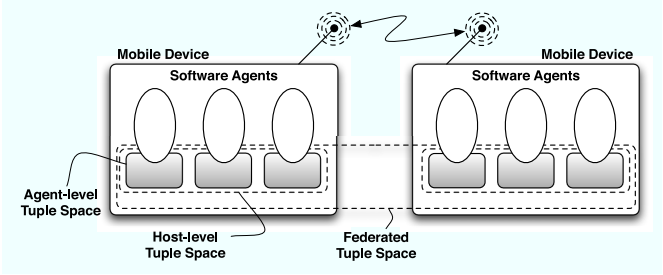


Figure 4: Agent-level, host-level and federated tuple spaces in LIME and TinyLIME.

Two agents are considered *connected* when either co-located on the same physical host or located on two different hosts able to communicate through the underlying network. The union of all tuple spaces of connected agents yields a *federated tuple space*, as illustrated in Figure 4, providing the illusion of a common memory space. This abstraction enables a form of communication decoupled in *time* and *space*: senders and receivers need not be active at the same time, nor do they need to know each others' identity or location.

With respect to LIME, TinyLIME brings motes into the picture: when a mote is in range of some mobile device, it is represented as another software agent *logically residing* on that mobile device. Therefore, it is *connected* to the other agents on this device, and becomes part of the host-level tuple space and hence the federated tuple space. It is the middleware that takes care of creating this abstraction, and of providing access to the tuples stored on the motes as if they were stored on any other mobile device. Clearly, this enables transparent access to the information located on the mobile devices as well as stored on the motes. This characteristic, along with the ability to restrict the scope of **in** and **rd** operations to a given agent or host, naturally provides *context-aware* operations. Indeed, one can query only the surrounding motes by simply limiting the scope of **rd** operations to her own host. Moreover, TinyLIME keeps the federated tuple space abstraction up to date across all devices as the underlying network topology changes, as illustrated in Figure 5.

As we show in the following section, many of the requirements arising in the implementation of "Save the Princess!" are easily solved by the combination of *i)* the data sharing abstraction that provides seamless access to a distributed and dynamically changing state described in terms of tuples, and *ii)* the ability to install reactions to fire in response to changes in this distributed state.

4. GAME DESIGN AND IMPLEMENTATION

To realize our pervasive game, we started by analyzing the functionality that must be available to each player. These are illustrated in Figure 6. The game is characterized by a set of *interactions* occurring either between the *players* and the *environment*, or *among players* themselves. The outcome of each interaction is reflected in a change in the game *state*, which is *distributed* across the players' devices and the motes. Therefore, we report on how we both represent the game state and realize the required interactions. Additionally, to better highlight the expressiveness of the TinyLIME programming model, we show representative code excerpts.

Representing the Game State. As all data is represented in TinyLIME as tuples, we make a straightforward mapping from all game elements into only six different tuple formats, as summarized in Figure 7. In our design, the tuples describing the game state can be located either on the mobile devices carried by the players or on the motes embedded in the environment. Where a tuple is stored implies certain semantics. For example, tuples stored on a mobile device correspond to objects currently *held* by the corresponding player. Conversely, the tuples stored on the motes represent objects *left* in the environment and available for pick up.

Interacting with Objects. To make the game more realistic, in-

Tuple Type	Format	Description
LocationTuple	⟨*id, name, description*⟩	Represents a location in the game. *name* represents the name of the location, e.g., "the entrance of the castle". *description* is a string shown to the user when she is logically in that particular location, e.g., to provide a narrative related to the location.
PlayerTuple	⟨*id, name, role, leader, strength, abilities, maxIngredients, maxObjects*⟩	Represents a player in the game. *name* represents her name, while *role* is a constant equal to either `WIZARD`, `DWARF` or `KNIGHT`. *leader* is a boolean flag stating whether this player is the leader of his team. *strength* reflects the current health status of the player. *abilities* encodes the player's current capabilities, while *maxIngredients* and *maxObjects* represent the maximum number of ingredients and objects a player can carry, respectively.
ObjectTuple	⟨*id, locationId, name, description, abilities, combinations*⟩	Represents an object the players can pick up or drop in a given location. *locationId* represents the id of the location where the object resides. When it is held by a player, this field is set to the player's id. *name* and *description* provide information similar to the same fields in a *LocationTuple*. *abilities* encodes the additional abilities a player gains in using this object. Correspondingly, *combinations* encodes the ids of the other objects this object can be combined with.
IngredientTuple	⟨*id, locationId, name, description, decay*⟩	Represents a magical ingredients for the final poison. *locationId*, *name* and *description* have the same meanings as in *ObjectTuple*. *decay* represents how deteriorated is the ingredient. When it reaches zero, the ingredient disappears and cannot be used.
EnemyTuple	⟨*id, locationId, name, type, strength, hidingLocationId*⟩	Represents an enemy in the game. *locationId* and *name* represent the place where the enemy currently resides and her name, respectively. *type* can be either `OGRE` or `SKELETON`, while *strength* represents the health status of the enemy. When it reaches zero, the enemy is defeated and the location she was hiding, represented by *hidingLocationId*, now becomes visible.
MessageTuple	⟨*id, locationId, senderName, receiverName, message*⟩	Represents a message in the game, and is used to let the players communicate with each other. *senderName* and *receiverName* represent the name of the sender and receiver players, respectively. They can also be left unspecified. Leaving *senderName* unspecified allows sending anonymous messages. Conversely, an unspecified *receiverName* enables the players to leave messages in the environment, making them available for pick up by any other player.

Figure 7: Tuple formats defined in "Save the Princess!".

```
private void dropObject(GameObject obj, GameLocation location, AgentLocation targetMote) {
  // Creates the tuple
  ITuple tuple = new Tuple().addActual(obj.getId()).addActual(location.getId()).addActual(obj.getName())
                            .addActual(obj.getDescription()).addActual(obj.getAbilities()
                            .addActual(obj.getCombinations());
  // Outputs the tuple
  lts.out(targetMote, tuple);
}
```

Figure 8: Code to drop an object in the game. `location` represents the current location where the player resides, and `targetMote` indicates the mote where the tuple will be stored.

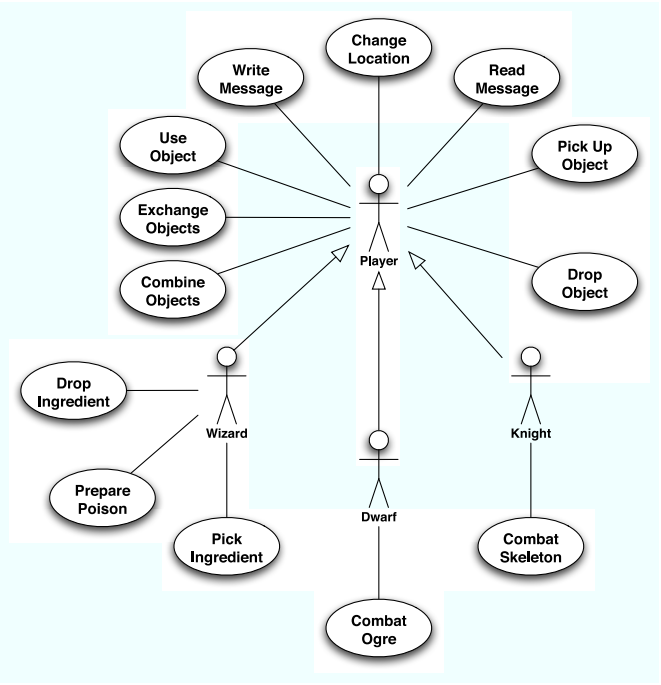


Figure 6: A subset of the use cases for "Save the Princess!".

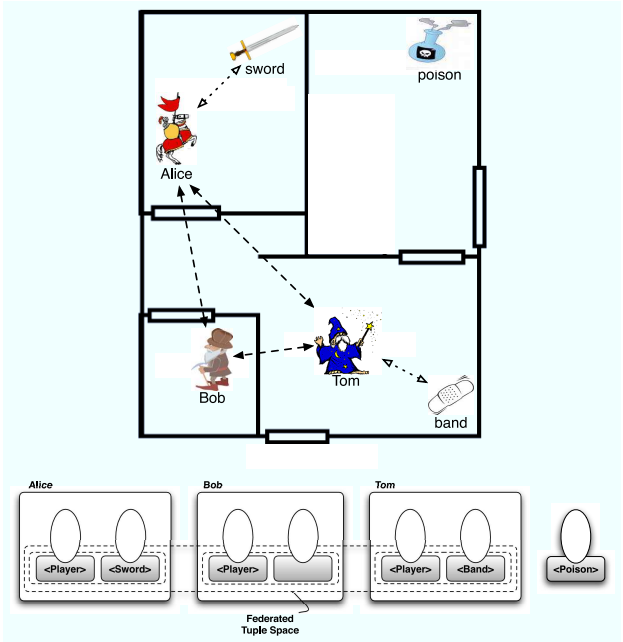


Figure 9: A possible situation in the game.

teractions are enabled or inhibited based on the current state. For instance, a player can only *use* an object if she is currently carrying it, i.e., the tuple representing the object is in the tuple space on her mobile device. More importantly, the operations to pick up or drop an object make sense only if the two parties involved are in *direct* communication. In TinyLIME, such object movement involves the corresponding movement of the tuple representing that object (an **in** followed by an **out**). By simply restricting the scope of the **in** operation to the local host (that includes the motes in range), the direct connectivity between the player and the mote is guaranteed. Figure 8 shows the code necessary to drop an object. Picking up an object and exchanging objects among players are similarly implemented.

At this point, it is interesting to look at Figure 9, where we illustrate the same situation as in Figure 5(b) in the context of "Save

```
private void locationManagementSetup(){
  // Creates the tuple
  ITuple tuple = new Tuple().addFormal(GameLocationIdentifier.class)
                            .addFormal(String.class)   // The name of the location
                            .addFormal(String.class);  // The associated description
  // Creates the template
  MoteLimeTemplate template = new MoteLimeTemplate(tuple);
  // Builds a location object in LIME corresponding the the local host
  HostLocation localHost = new HostLocation(
              new LimeServerID(InetAddress.getLocalHost(), agent.getMgr().getPort()));
  // Creates a reaction localized to the local host
  Reaction r = new LocalizedReaction(localHost, new AgentLocation(agent.getMgr().getID()),
              new LocationChangeListener(), template, Reaction.ONCEPERTUPLE);
  // Registers the reaction
  lts.addWeakReaction(new Reaction[]{r});
}
```

Figure 10: Installing a reaction for location tuples. `lts` is a reference to an instance of `MoteLimeTupleSpace` providing access to the federated tuple space. (This is actual code, only exception blocks are omitted for readability).

```
public class LocationChangeListener implements ReactionListener {
  public LocationChangeListener(GameEngine gameEngine){
    // ...
  }
  public void reactsTo(ReactionEvent mre) {
    LocationTuple newLocation = (LocationTuple) mre.getEventTuple();
    gameEngine.notifyLocationChange(newLocation);
  }
}
```

Figure 11: Handling a location change after the reaction installed in Figure 10 fired. The location tuple is passed to the game engine to process a possible change of location.

the Princess!". The bottom of Figure 9 shows how TinyLIME renders the situation depicted at the top of the same figure. The mote storing the tuple representing the poison is currently not in range of any player. As such, the poison cannot be picked up. On the other hand, Alice is in range of a sword. Therefore, she can pick it up. If she does so, this will be reflected by moving the tuple representing the sword from the mote to Alice's device. At the bottom of Figure 9 Bob is shown in range of a mote, but this mote does not store any tuples. Therefore, Bob cannot pick up any objects at his current location, but he could drop something there.

Game Location Changes. Each player has an associated virtual location she is residing in. However, this virtual location is tied to the physical world, as it is inferred from the motes in range storing tuples of type *LocationTuple*. This avoids the use of GPS receivers or more complex location mechanisms.

In particular, players infer their location based on the location tuples they have been in direct contact with. The most recently seen location tuple serves as the player's location, therefore even if a player is not in range of a mote with a location tuple, it retains the most recent (virtual) location. Keeping this information up to date is as straightforward as registering a reaction for tuples of type *LocationTuple* with the scope of the operation restricted to the local host. Figure 10 shows the installation of this reaction while Figure 11 shows the reaction listener code that will be called when the reaction fires. How the game handles the reaction is independent from the code that fires the reaction.

Finally, the game does not always report a new location up to the player graphical interface when the aforementioned reaction fires. If an enemy is present at the newly discovered location, the player is prevented from entering the location unless she defeats the enemy. Therefore, the presence of the enemy is notified instead of the new location. To detect if an enemy is present, when the location reaction fires a **rd** operation is issued to look for local enemy tuples. If no tuple is returned, the player is notified about the new location and any objects and players present in it.

Player-to-Player Interaction. Players interact among themselves either to *exchange objects* or to *communicate*. Similarly to the process for picking up and dropping objects, two players are allowed to exchange an object only when their mobile devices are within communication range. When this condition is true, object movement is reflected in the movement of its corresponding tuple from one player's mobile device to the other, again using **in** and **out** operations.

Regarding communication, TinyLIME is used to enable both *transient* and *persistent* forms of communication. The former is supported as long as communication is available. In this case, the message tuple is directly output to the tuple space of the receiving player. A reaction for message tuples restricted to the local tuple space fires, removing the tuples from the tuple space and displaying the message content to the player.

In contrast, we also allow players to *leave* messages in the environment, e.g., stored by issuing an **out** operation to that mote. This enables a form of persistent communication, decoupled in time. Messages can be left in the environment for an unbounded amount of time. When a player moves to a new location, a reaction notifies her about the messages found in it. The player can read the messages or even remove them, making them unavailable to future players passing through the same location.

5. TINYLIME FOR PERVASIVE GAMES

The version of TinyLIME we used for developing our game differs from the original described in [4]. Indeed, data collection in wireless sensor networks—the original TinyLIME target domain—does not require applications to request data storage on the motes. Therefore, the original version allowed only operations on the mote tuple spaces that did not change their content (namely **rd** and reactions). In fact, inside the implementation tuples were never even physically present on the motes; they were generated on the fly by the middleware when data was requested. Instead, in the context of this paper the capability to store and manipulate data on the motes is vital to manage the virtual objects. Therefore, we extended Tiny-

LIME to allow storage of arbitrary tuple formats and to enable **out** and **in** operations to manipulate the tuple space.

We faced several options to include these extensions. On one hand, the motes could be considered as any other host, and all the operations would be implemented by exchanging messages from the mobile device to the motes. For instance, an $\mathbf{in}(p)$ operation could be implemented by sending a message containing the pattern p to the mote, this pattern would be matched against the stored tuples, and possibly remove one of them. This approach would require an explicit message for *each* operation involving the motes, as well as a suitable encoding in nesC of the matching semantics.

While using a message for each operation is possible, it is likely to be too expensive in terms of battery consumption on the motes. Further, an expressive matching semantics involving comparisons among complex data types [21] would be prohibitive on a resource-poor device such as the Crossbow MICA2DOT [1]. In addition, the limited storage space on motes constrains the number of tuples that can be hosted. Finally, operations that do not change the contents of the tuple space (e.g., **rd** operations) are likely to constitute the vast majority of the processing. For these reasons, we chose to *replicate* the tuples stored on a mote on each mobile device in range of that mote, to perform all the processing on the local replica owned by the mobile device, and to reflect the changes (if any) back to the mote. This requires only 1459 lines of non-commented nesC code. Further, because the implementation is in nesC [9], a rather low level programming language whose expressive power is comparable to that of ANSI C, porting TinyLIME to other embedded devices should be straightforward. Note also that this replication may generate consistency issues when more than one mobile device is in range of the same mote, as discussed later.

Replication and Write Operations. Tuples are stored on the motes as *opaque* types, i.e., a sequence of bits associated to a unique identifier. As soon as a mote enters the range of a mobile device, it communicates all the tuples it currently stores to the mobile device. These can be packed in a single physical message, optimizing the communication overhead. The mobile device decodes the sequence of bits received and creates actual tuples. These are inserted in a tuple space attached to a local software agent representing the mote in range. All read-only operations are performed on this tuple space, including explicit **rd** operations as well as the processing needed to possibly fire reactions matching the tuples on the mote.

We use explicit acknowledgments to make write operations reliable. In particular, a **out** operation is performed by having the mobile device encode the tuple and generate a unique identifier for it before shipping the data to the target mote. The mote replies with an acknowledgment after it has successfully stored the new tuple. At this time, the change is reflected in the local replica of the mote tuples. Similarly, to implement **in** operations, the mobile device first matches the pattern against the local replica of the tuples stored on the mote. If a matching tuple is found, its identifier is communicated to the mote where the tuple is removed, and an acknowledgment sent. At this point, the tuple is removed from the local replica as well.

Consistency Handling. Consistency issues may arise if more than one mobile device is in range of the same mote. Two mechanisms are used to manage this problem in our modified version of TinyLIME. First, each mote inserts the tuple it added or removed[1] in the acknowledgment it sends to the requesting mobile device. If other mobile devices are in range of that mote, they will also overhear the same acknowledgment. This way, all devices have all the necessary information to reflect the operation also in their local replicas. Secondly, each mote inserts the identifier(s) of the tuples it currently stores in the message used to advertise its presence. This way, each mobile station in range can recognize potential mismatches between the local replica and the tuples on the motes. In case of an **in** operation not yet reflected in the tuple space, the mobile device will simply remove a tuple from the local replica, without the need for further communication. Conversely, in the case of a missing tuple, the mobile device can explicitly retrieve it from the mote.

[1] In the case of an **in** operation the tuple identifier is sufficient.

6. RELATED WORK

While the use of augmented artifacts to embed computing within the environment is not novel [14, 23], to the best of our knowledge we are the first to use environmentally-immersed devices such as the Crossbow's MICA2DOT motes to store game information. In contrast to their use in sensor networks, we leverage their computing capabilities to represent virtual objects in a pervasive game, and exploit their small size to allow them to "disappear" [24] in the physical world. In [17] motes are attached to the players of a centralized pervasive game, but their storage and computing capabilities are not exploited, as they are used only to gather light and acceleration data. Conversely, the work in [25] provides only the software architecture to manage sensed information, still considering the sensors simply as data sources, and does not explore their actual use in developing pervasive games.

The use of wireless networks in pervasive games has been explored in several proposals, e.g., [16]. In particular, mobile ad-hoc networks offer a promising infrastructure for building games augmented with physical interactions and movements [22]. In this work, we further push this approach by proposing a two-tier architecture in which the ad-hoc network spans both the mobile devices of the players and the motes deployed in the environment.

Existing proposals in the field of middleware for game development have mostly addressed massive, multiplayer, online games, and concentrate on communication aspects. For instance, the work in [7] proposes a content-based, publish-subscribe [6] communication infrastructure to face the dynamic aspects stemming from players joining or leaving unpredictably. Similarly, the proposal in [29] discusses a communication infrastructure based on grouping players in clusters to achieve load-balanced event delivery. Real-time aspects are instead taken into account in [12], where the authors illustrate a system and algorithms to achieve fair message delivery in multiplayer games based on a client-server infrastructure. In TinyLIME, the scenario is radically different, as we target wireless ad-hoc networks with dynamic topologies instead of a quasi-static, wired network infrastructure. More importantly, we provide not only a communication infrastructure, but the abstractions of TinyLIME also enable state-full coordination mechanisms to govern the (distributed) game state and its evolution.

Finally, middleware for pervasive computing in general is an active area of research [19]. Most of the solutions in this field focus on exporting devices to the applications in terms of available services, e.g., as in [11]. Therefore, discovery and composition of services constitute the core of the abstractions offered to programmers. Conversely, in TinyLIME we take a data-centric approach, making data (i.e., tuples) the main tool developers exploit to program their applications. As we have demonstrated in this paper, this actually constitutes a natural abstraction for game developers.

7. CONCLUSION

We presented the design and implementation of a pervasive game, supporting two contributions. On one hand, our game makes an

original and unprecedented use of wireless sensor networks, treating them as tiny computing devices maintaining the game representation of virtual objects, characters, and locations. On the other hand, by analyzing the game design and implementation using TinyLIME, we show that this middleware naturally and effectively supports the development of pervasive games.

Acknowledgements. The authors wish to thank Alessandro Camerone, Simone Campanoni, Fabio Castignetti, and Emanuela Ceroni for their contribution to the implementation of "Save the Princess!" and the extensions to TinyLIME. The work described in this paper was partially supported by the European Community under the IST-004536 RUNES project and by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation under grant number 5005-67322.

8. REFERENCES

[1] Crossbow Technology Inc. `www.xbow.com`.

[2] I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. A survey on sensor networks. *IEEE Communication Mag.*, 40(8):102–114, 2002.

[3] A. D. Cheok, S. W. Fong, K. H. Goh, X. Yang, W. Liu, and F. Farzbiz. Human pacman: a sensing-based mobile entertainment system with ubiquitous computing and tangible interaction. In *Proc. of the 2^{nd} Workshop on Network and system support for games (NETGAMES)*, 2003.

[4] C. Curino, M. Giani, M. Giorgetta, A. Giusti, A. L. Murphy, and G. P. Picco. Mobile data collection in sensor networks: The TINYLIME Middleware. *Elsevier Pervasive and Mobile Computing Journal*, 4(1):446–469, Dec. 2005.

[5] C. Curino, M. Giani, M. Giorgetta, A. Giusti, A. L. Murphy, and G. P. Picco. TINYLIME: Bridging mobile and sensor networks through middleware. In *Proc. of the 3^{rd} IEEE Int. Conf. on Pervasive Computing and Communications (PerCom)*, 2005.

[6] P. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 2(35):114–131, June 2003.

[7] S. Fiedler, M. Wallner, and M. Weber. A communication architecture for massive multiplayer games. In *Proc. of the 1^{st} Workshop on Network and system support for games (NETGAMES)*, 2002.

[8] M. Flintham et al. Where on-line meets on the streets: experiences with mobile mixed reality games. In *Proc. of the SIGCHI Conf. on Human factors in computing systems (CHI)*, 2003.

[9] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'03)*, 2003.

[10] D. Gelernter. Generative communication in Linda. *ACM Computing Surveys*, 7(1):80–112, January 1985.

[11] R. Grimm et al. System support for pervasive applications. *ACM Trans. Comput. Syst.*, 22(4):421–486, 2004.

[12] K. Guo, S. Mukherjee, S. Rangarajan, and S. Paul. A fair message exchange framework for distributed multi-player games. In *Proc. of the 2^{nd} Workshop on Network and system support for games (NETGAMES)*, 2003.

[13] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proc. of the 9^{nt} Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.

[14] A. Kameas, S. Bellis, I. Mavrommati, K. Delaney, M. Colley, and A. Pounds-Cornish. An architecture that treats everyday objects as communicating tangible components. In *Proc. of the 1^{st} IEEE Int. Conf. on Pervasive Computing and Communications (PerCom)*, 2003.

[15] C. Magerkurth, A. D. Cheok, R. L. Mandryk, and T. Nilsen. Pervasive games: bringing computer entertainment back to the real world. *Comput. Entertain.*, 3(3), 2005.

[16] K. Mitchell, D. McCaffery, G. Metaxas, J. Finney, S. Schmid, and A. Scott. Six in the city: introducing Real Tournament—a mobile IPv6 based context-aware multiplayer game. In *Proc. of the 2^{nd} Workshop on Network and system support for games (NETGAMES)*, 2003.

[17] S. N. I. Mount, E. I. Gaura, and R. M. Newman. Sensorium games: usability considerations for pervasive gaming. In *Proc. of the 23^{rd} Int. conference on Design of communication*, 2005.

[18] A. L. Murphy, G. P. Picco, and G.-C. Roman. LIME: A Coordination Model and Middleware Supporting Mobility of Hosts and Agents. *ACM Transactions on Software Engineering and Methodology*, 15:279–328, 2006.

[19] E. Niemela and J. Latvakoski. Survey of requirements and solutions for ubiquitous software. In *Proc. of the 3^{rd} Int. Conf. on Mobile and ubiquitous multimedia (MUM04)*, 2004.

[20] C. E. Perkins. *Ad hoc networking*. Addison Wesley, 2001.

[21] G. P. Picco, D. Balzarotti, and P. Costa. LIGHTS: A lightweight, customizable tuple space supporting context-aware applications. In *Proc. of the 20^{th} ACM Symposium on Applied Computing (SAC)*, 2005.

[22] S. M. Riera, O. Wellnitz, and L. Wolf. A zone-based gaming architecture for ad-hoc networks. In *Proc. of the 2^{nd} Workshop on Network and system support for games (NETGAMES)*, 2003.

[23] M. Roj. Smart artifacts as key component of pervasive games. In *Proc. of the 2^{n} Int. Workshop on Pervasive Games Applications (PERGAMES05)*, May 2005.

[24] N. Streitz and P. Nixon. The disappearing computer. *Commun. ACM*, 48(3), 2005.

[25] R. Suomela, K. Koskinen, and K. Heikkinen. Rapid prototyping of location-based games with the multi-user publishin environment application platform. In *Proc. fo the IEEE Int. Workshop on Intelligent Environments*, June 2005.

[26] B. Thomas, B. Close, J. Donoghue, J. Squires, P. D. Bondi, and W. Piekarski. First Person Indoor/Outdoor Augmented Reality Application: ARQuake. *Personal Ubiquitous Comput.*, 6(1), 2002.

[27] S. P. Walz et al. Cell spell-casting: Designing a locative and gesture recognition multiplayer smartphone game for tourists. In *Proc. of the 3^{rd} Int. Workshop on Pervasive Gaming Applications at PERVASIVE 2006*, May 2006.

[28] M. Weiser. Ubiquitous computing. *IEEE Computer*, 26:71–72, 1993.

[29] S. Yamamoto, Y. Murata, K. Yasumoto, and M. Ito. A distributed event delivery method with load balancing for mmorpg. In *Proc. of the 4^{th} Workshop on Network and system support for games (NETGAMES)*, 2005.