# The .NET CF Implementation of GecGo
# - A middleware for multihop ad-hoc networks -

Peter Sturm, Daniel Fischer, Volker Fusenig, Thomas Scherer
University of Trier
Department of Computer Science
D-54286 Trier, Germany
{sturm,fischer,fusenig,scherer}@syssoft.uni-trier.de

**ABSTRACT**

The goal of the GecGo middleware is to provide all the services required by self-organizing distributed applications running on multihop ad-hoc networks. Because of the frequent as well as unreliable and anonymous communication between accidental neighbors observed in these networks, applications have to adapt continuously to changes in the mobile environment and the GecGo middleware offers the required tight coupling. Additionally, GecGo addresses specifically the issue of "en passant" communication, where moving neighbor devices may interact only for short periods of time. In this paper, the architecture and basic concepts of the GecGo middleware are discussed and a prototype implementation of GecGo using the Microsoft Windows CE 4.2 .NET operating system for mobile devices and the .NET Compact Framework is presented.

**Keywords**
Middleware, Mobile Networks, Multihop Ad-Hoc Networks (MANET), Mobile Applications, Self-Organization

## 1. INTRODUCTION

The emerging capabilities of modern mobile devices with respect to CPU power, wireless communication facilities, and battery capacity are the foundation of future multihop ad-hoc networks. The frequent as well as unreliable and anonymous communication between accidental neighbors observed in these mobile networks makes their successful deployment a challenging task. With the absence of any reliable backbone network, all mobile devices have to participate altruistically in a distributed execution environment with some kind of epidemic message delivery. Self-organization is the most promising design principle in order to manage these networks successfully and efficiently. As a consequence, any decision of a mobile device must be based on local as well as on current neighborhood knowledge and common goals must be achieved by means of synergy.

Any fundamental communication pattern in such a network exhibits an *en passant* characteristic. Two devices are within communication range for a short period of time and while they pass each other, they might cooperate and exchange certain data. In most cases, being within communication range with a given device is purely accidental and the probability to meet this device again in the near future is fairly low. During this en passant communication, applications and middleware must agree fast on which entities should change the hosting device in order to get closer to their final destination. The required decisions depend on a number of factors, among others the importance of the moving entity, the size of the entity compared to an estimation of the remaining interaction period, and the future direction of the neighbor with respect to the final destination.

These stringent conditions for distributed applications in multihop ad-hoc networks aggravate the need for the continuous adaption to a dynamically changing environment. As a consequence, this requires a very tight coupling between the mobile applications and the middleware. Many high-level mechanisms that are common in traditional system software and middleware that trade transparency vs. performance are therefore inadequate. The goal of the GecGo middleware (Geographic Gizmos) is to offer this tight interaction with application components and to provide all the necessary services required by self-organizing systems running on multihop ad-hoc networks. The first prototype of this middleware has been implemented on Microsoft Windows CE 4.2 .NET using the .NET Compact Framework.

In the next section, the fundamental concepts and the basic functionality of the GecGo middleware are

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, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*.NET Technologies'2004 workshop proceedings,*
*ISBN 80-903100-4-4*
Copyright UNION Agency – Science Press, Plzen, Czech Republic

introduced. The functionality of the GecGo middleware is discussed in section 3. The architecture and implementation issues of the .NET prototype are discussed in section 4. The paper ends with an overview on related work and a conclusion.

## 2. GECGO CONCEPTS

The conceptional structure of the middleware and its four basic abstractions are depicted in figure 1. Any mobile or stationary device participating in the GecGo runtime environment is represented by a `DeviceGizmo` and the code of GecGo applications is derived from the base class `CodeGizmo`. Every code has its residence in form of a device. Depending on the distributed execution model, this residence remains fixed or it might change over time (mobile agents). For application code with a fixed residence, GecGo provides the abstraction of mobile state (`StateGizmo`) that might change the hosting device instead of the code. Since end-to-end messages between devices may remain on a device for a longer period of time in case no suitable neighbor is found, they also exhibit a more state-like nature. As a consequence, messages are represented in GecGo as special cases of a `StateGizmo`.

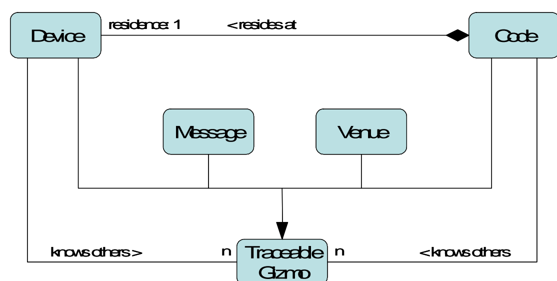


*Figure 1: Main GecGo Abstractions*

The fourth abstraction is defined by the `VenueGizmo` which ties a logical place or an event to a well-defined set of geographic coordinates and time slots, e.g. a several week long lecture on distributed systems with changing rooms and time slots. `VenueGizmos` are virtual in the sense, that they bear no computational resources per se. Instead they rely on the devices that are within a given distance from the venue center. Entities with a venue as their destination will first try to reach a device at the venue. As long as they have no other destination, they will try to remain at the venue possibly by changing the hosting device.

### GeoTraces

All major abstractions in GecGo are derived from a fundamental data type `TraceableGizmo` (see figure 2). Any subtype of this class is traceable in time and space by means of a `GeoTrace`. These traces keep accounts on events in the past, they reflect the present situation, and they store estimates about future events. The actual information stored in the trace of a gizmo is defined by its type and consists of a set of so-called `Gepots` (pieces of time and geographic data). Also the depth and the level of detail of the `GeoTrace` depends on resource considerations and the actual type of gizmo. For example, `DeviceGizmos` keep track about where they have been in the past, at what time as well as why and they may also store information about previous neighbor devices. The present informs about the current position of the device and the actual neighborhood. The future trace might contain estimates where the device will be in the future, e.g. students will be in certain future lectures with a high probability.

Traces of `StateGizmos` will be more resource-limited. They will store at least the final destination as part of the future trace. `VenueGizmos` are even more restricted, since they represent only virtual entities within the GecGo environment. As such, the trace of the venue is identical to the time schedule of the event associated with this venue. Additional data that might be important to run the venue must be stored by the hosting devices that are currently within the vicinity of the venue center.

All devices are required to update their traces continuously over time. With the goal to keep the number of Gepots in the past to a reasonable minimum, the information stored in the present of the trace will be shifted into the past, e.g. when a mobile device starts moving again. The traces of devices are also the primary source for changes in the traces of other currently hosted state and code gizmos.

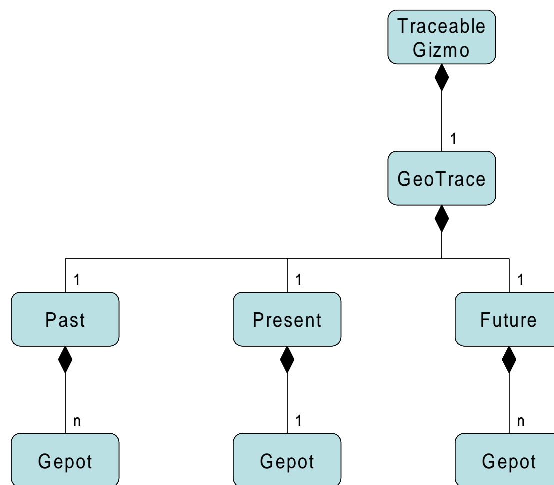


*Figure 2: Basic Type "Traceable Gizmo"*

From a conceptual point of view, the main function of the GecGo middleware enables traceable gizmos to move towards new destinations. The most common type of movement allows for mobile state to reach a given mobile device or to get into the vicinity of a certain venue, e.g. to implement the marketplace communication pattern for multihop ad-hoc networks as presented in [2]. In this context, a marketplace is a application-defined geographic area with an expected

high density of mobile devices (e.g. a lecture hall). Entities of a mobile application move between mobile devices of the users and the marketplace using a geographic routing protocol. By concentrating applications on specific geographic areas, marketplaces increase the probability that corresponding entities of the application get in contact with each other.

Covered by the concepts of GecGo are also the movement of mobile devices to reach a given venue (the special case of a navigation system, of course with the physical help of the human device owner) and movement of mobile code between devices as a means to implement mobile agent systems.

## 3. GECGO ARCHITECTURE

The basic architecture of the GecGo middleware consists of two gizmo management domains (see also figure 3):

- the `Lobby` for all the gizmos that are in transit and haven't reached their final destination yet
- the `Residence`, with gizmos that are intended to stay at this device for a longer period of time

Movement of gizmos from the lobby to the residence and vice versa will be performed with the aid of the porter service. Primarily, the porter is responsible for securing the identity of incoming gizmos and for providing the resources requested.

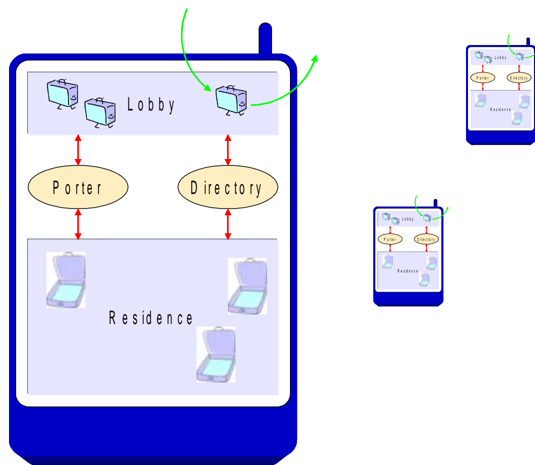


*Figure 3: GecGo Device Architecture*

### The Directory Service

A central directory service keeps track on any changes in both management domains. The directory itself has a hierarchical structure with leaves at the gizmo level (see figure 4). Applications may query the directory with wildcards to locate the required information. Most of the attributes of a gizmo entry are application-specific. For this purpose, the directory allows the definition of arbitrary key/value pairs as part of a gizmo leave. Additional attributes inside the directory tree are defined by the GecGo middleware and primarily serve infrastructure purposes such as the number of gizmos actually stored in the lobby or the list of neighbor devices moving in a southern direction.

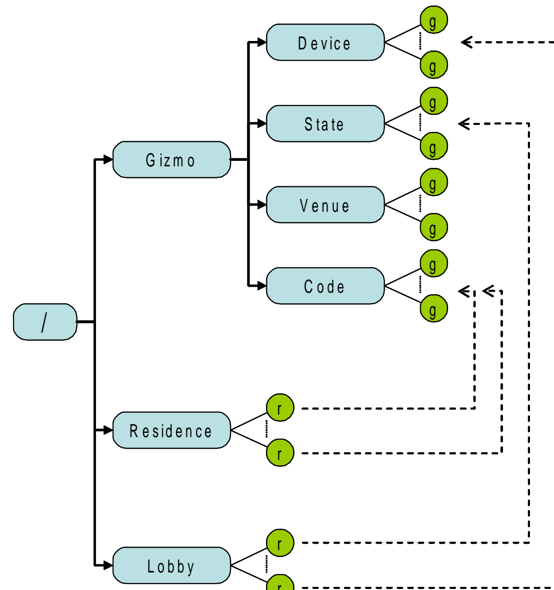


*Figure 4: The Directory Tree*

As depicted in figure 4, the directory tree consists of three major branches. The branch named Gizmo contains all the information and attributes about any gizmo located on the given mobile device. Additionally, entries about specific gizmo types that are known by a mobile application but where no instance is available on the device yet can be inserted by application components. In conjunction with the asynchronous notification mechanism described below, applications can react appropriately on future events.

The remaining two branches below the root of the tree keep track on the gizmos located in the residence or the lobby. Primarily, these branches contain references to specific gizmos that are part of these domains. The residence branch holds references to code gizmos that are installed on the mobile device or that entered the device via the porter in case of a mobile agent environment. The lobby has references to the gizmos awaiting a device change in order to reach their final destination. Most of these gizmos are of type `StateGizmo`, although again in a mobile agent environment, the lobby could also be occupied by code gizmos that might change the host or enter the residence through the porter. Also part of the lobby branch are references to device gizmos that represent the current neighborhood of the mobile device. This information about the current neighborhood is also available as part of the device' present `GeoTrace`.

Gizmos in the lobby and the residence may query for the existence of certain gizmos and they may register a callback to be informed about specific events. For this purpose, every gizmo has an application defined unique name that serves as the key for the directory service. Possible events supported by the middleware are:

- arrival of a new gizmo with a specified type in the lobby of a device
- departure of a gizmo from the lobby
- movement of gizmos between the lobby and the residence
- instantiation and deletion of new gizmos by means of code gizmos in the residence

## Communication with direct Neighbors

Singlehop communication services for gizmos with devices in the immediate neighborhood are provided by the middleware kernel. Therefore the middleware has to know all available devices within singlehop communication range. Information about these devices are periodically updated in the lobby branch of the directory tree as references to device gizmos. These entries store the following attributes:

- an unique device id, which is used to resolve the current ip address of the device
- the port which is used for gizmo transmissions
- a flag indicating if the device currently accepts incoming gizmos
- the future GeoTrace of the mobile device in order to ease the decision which gizmos to move during the en passant communication

The required information is exchanged among mobile devices by broadcasting a beacon in regular time intervals using an UDP based communication protocol. These beacons contain all the aforementioned information to ease gizmo exchange. The protocol is also being used to transmit other kinds of messages such as termination signals or requesting a certain gizmo. With the device information—containing ip address and port of a potential communication partner—the middleware is able to establish a TCP connection to transfer a gizmo.

## Mobile State vs. Mobile Code

A central decision in mobile ad-hoc networks addresses the issues on mobile code vs. mobile state. Mobile agents are an interesting technology for wireless and mobile networks with far reaching implications on system security and code integrity. The GecGo middleware covers mobile agents in its model by accepting a changing residence for mobile code. This functionality is currently not part of the .NET implementation of the GecGo middleware platform. Besides technical reasons, this decision is primarily driven by a number of unsolved problems with respect to the limited resources on a mobile device, the larger amount of data required to move mobile code including its execution state transparently from one device to another, and the need to authenticate and secure code execution.

Instead of mobile code, the GecGo middleware actually offers so-called *mobile state*, which requires the application components to cooperate non-transparently in packaging and unpacking execution state into and from mobile state gizmos. The middleware offers several functions and services to ease this task for the application code. In contrast to mobile code, applications must be installed explicitly by the user of a device, before state gizmos for a given application can be received and processed on their final destination. Of course no application code must be installed on devices that are only intermediate hosts for state gizmos.

## An Routing Gizmo Example

To illustrate the dynamics inside the GecGo middleware, an example for a multihop routing service is elaborated in more detail. In this scenario, a routing gizmo—a subclass of a code gizmo—implementing a version of a geographic routing protocol is available as a pre-installed GecGo application. In this constellation, the routing gizmo is responsible for the end-to-end communication in the ad-hoc network on the basis of exchanging state gizmos between neighboring devices as provided my the middleware.

The routing gizmo registers a callback with the event that will be invoked in case a new neighbor device is within communication range. The event causes the routing gizmo to examine the beacon information of this neighbor in order to decide which state gizmo to transfer. As mentioned above, part of the beacon information are geographical coordinates about future positions of the neighbor device. This position information is the primary source for the routing decision, e.g. if a venue in the future trace of the neighbor device is identical to the future venue of a candidate state gizmo or if the trajectory of the neighbor device is targeting towards the destination of the state gizmo.

In a first scenario we assume that a state gizmo *g*, which is currently part of the lobby on a device *d1*, wants to reach a venue gizmo *v*. Each time such a new state gizmo with different destination enters the lobby of a device, the enter-event triggers the execution of specific callback in the routing gizmo in order to store its destination venue. Suppose now that another device *d2* is passing by *d1*. In that case the routing gizmo decides with respect to the used routing protocol (e.g. greedy routing) whether *g* moves from *d1* to *d2* or not. This procedure is repeated on different devices until *g* arrives on a device close to its destination venue.

In another routing scenario, gizmos residing on two different mobile devices want to exchange information directly, for example a mobile application on device *d1* wants to send a state gizmo *g* to another application on a specific device *d2*. Since a routing path from *d1* to *d2* among multiple mobile devices

cannot be maintained because of the dynamics in multihop ad-hoc networks, *d1* must determine the position of *d2* in some other way. One solution to this problem is, to hash the identification of *d2* to a specific geographic position within a given area (e.g. the university campus). This hash function is identical on every participating device and consists of a classical cryptographic hash to achieve a statistically unified distribution and a subsequent mapping of this randomized identifier to a geographic position. The calculated geographic position will be the target for the state gizmo in the same manner as in the example above. In return the device *d2* itself is able to compute the same geographic position and it might issue additional requests to collect the information or to periodically send updates about its position to this venue.

## 4. .NET IMPLEMENTATION

A first prototype version of GecGo has been implemented using the Microsoft Windows CE 4.2 .NET operation system and the .NET Compact Framework. The middleware has been written in C#. At the time of writing, the middleware has a size of approximately 3000 lines of C# source code. The GecGo executable itself has a size of 84 KBytes at runtime without any mobile applications installed. The target mobile devices are Compaq IPAQs H5550 with 128 MByte main memory and integrated WLAN communication facilities. The middleware can also be executed on ordinary notebooks supporting the full version of the .NET framework. The porter service as described in section 3 is not part of the current implementation. The main reason for this is the concentration on mobile state instead of mobile code. As a consequence, all mobile applications are currently installed explicitly on any participating mobile device with no need for the porter services.

The GecGo middleware architecture consists of five major classes:

- `Middleware`: This class serves as the main access point for mobile applications. The class also handles incoming state gizmos and enables the execution of new applications.
- `Beacon`: The UDP beacon required for the discovery of devices within communication range is sent periodically by an instance of this class.
- `UDPListener`: Primarily, this class handles incoming beacons of other devices. It also acts on the receipt of UDP-based requests by neighbor devices to prepare for the transmission of another gizmo.
- `TCPServer`: This class is used on the receiving side to transmit complete gizmos from one device to another. The device willing to send a gizmo takes up the client role.
- `GecGoDirectory`: This class defines the interface to all directory-related functions of GecGo.

The single instance of class `Middleware` implements the graphical user interface of GecGo. The user can browse the mobile applications installed on the device and select individual applications to be executed. For this purpose, the Middleware maintains a hash table to derive the reference to the corresponding assembly via its name. Each time an application is started, the entry point of the assembly—a public static method called `run(IMiddleware mid)`—is executed. In this call, the argument `mid` of type `IMiddleware` defines all the functions that are available to mobile applications. In the prototype version of GecGo, the following methods and properties are provided:

- `SendToAllNeighbors(Gizmo g)`
- `SendGizmo(Gizmo g, Device r)`
- `RegisterApplication(Assembly a)`
- `IGecGoDirectory gd`

The property `gd` returns a reference to the directory service of the middleware through the interface `IGecGoDirectory`. This interface encapsulates all the directory functions available to the mobile application. Among others, the directory service currently defines the following methods:

- `InsertGizmo(Gizmo g)`
- `DeleteGizmo(Gizmo g)`
- `RegisterEvent`<br>`(Delegate f, string regExp)`
- `UnregisterEvent(Delegate f)`
- `GetGizmos(string regExp)`

The delegate mechanism of C# is used to implement the asynchronous callback mechanism of the GecGo directory service. For example, if a gizmo inside the residence wants to be notified upon the arrival of gizmos of a given type T in the lobby, it registers a delegate with the event `/Lobby/T/<Enter>` and the middleware will call back each time such a gizmo enters the device. The .NET events may also be used to implement asynchronous notifications between different gizmos.

The different communication tasks of the middleware are handled by 3 dedicated threads, which are created through instances of the classes `Beacon`, `TCPServer`, and `UDPListener`. Any device maintains information about its current neighborhood through the listening UDP thread which receives any beacon messages issued by the Beacon thread of nearby devices. The detailed information about the neighbor will be continuously reflected by the structure of the directory service and may—for example—trigger the transmission of state gizmos in the lobby of a device. In this case, an UDP request is sent to the potential next host. If this device is willing to act as a host, a TCP connection will be established and the selected gizmo will be transmitted. The decision to rely on TCP connections for the transmission of gizmos even if the size of the gizmo would fit into a single WLAN frame only holds for the prototype version of GecGo.

While implementing the thread-based communication part of the middleware, several limitations of the .NET compact framework became obvious. Several management functions of the full .NET framework such as asynchronous thread termination are missed in the compact version of .NET. As a consequence, more complex synchronization techniques must be implemented to manage the active number of threads. Also the interaction between beaconing and dealing with the receipt of UDP messages would benefit from additional functionality around UDP sockets, e.g. to add a time-out value to a blocking receive on an UDP socket.

Before sending, any gizmos are serialized. The binary format of a serialized gizmo follows the TLV principle (Type, Length, Value). In the current version of GecGo it is assumed that a gizmo is defined by the values of the non-static class members, so that there are no interface implementations needed to marshall objects. Thus, the serialized form is a sequence of these values. Each member is described by a type, a name and a value triple. Non-primitive values are marshalled recursively in the same way and arrays are characterized by the enumeration of their values. By using the reflection mechanisms of the .NET Compact Framework the member informations and values are queried independently from their visibility. The reason for the explicit implementation of a gizmo serialization is the missing support of the `BinaryFormatter` and the `SoapFormatter` classes of the .NET framework in the compact version. The main reason for this were size and performance considerations which might hold for today's mobile devices. But in anticipation of future heterogeneous and mobile computing environments, there will be a increasing need for a standardized serialization mechanism besides accessing web services.

## Threads in GecGo

A first set of experiments was targeted towards the Microsoft system software in order to determine the costs induced in a multi-threaded implementation of the middleware running on top of Windows. Observations at the beginning of the implementation phase lead to the initial decision, to implement a first middleware version with a limited number of threads only, because thread instantiations, deletions, and context switches appeared to be to costly on the IPAQ devices. In order to verify this observation, a simple C# test program has been implemented which instantiates a variable number of threads. Each of these threads yields the CPU in a while loop for a given period of time (2 minutes). The calculated time for a single context switch is depicted in figure 5 for 2 to 256 threads in a single address space (application domain). The same source code has been compiled for 2 different release platforms: (1) a Windows console application with the full support of the .NET foundation classes (Version 1.1) and (2) a smart device application using the .NET compact framework only. Both executables have been executed on a 1900 MHz mobile Intel Pentium 4. As the graphs in figure 5 indicate, the time for a single context switch increases slowly with the number of threads, starting with 937 ns (.NET CF executable) and 915 ns (full .NET executable) for two threads. The absolute number for 256 threads are 3361 ns (.NET CF executable) and 3438 ns (full .NET executable).

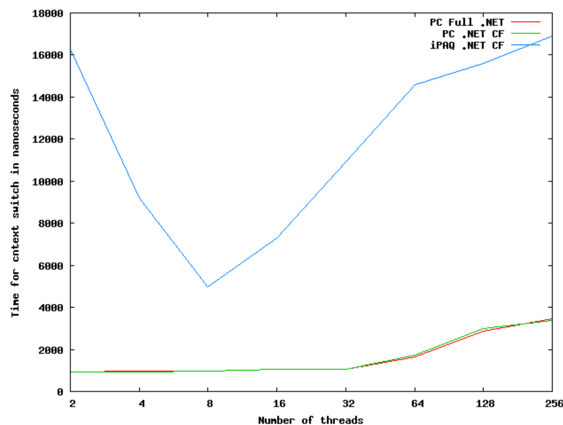


*Figure 5: Duration of context switches*

The absolute times for the iPAQ .NET CF executable running on a 400 MHz Intel Xscale processor are 16.28 microseconds for 2 threads and again 16.889 microseconds for 256 threads. A little surprising was a significant minimum of 4.99 microseconds for 8 threads. Provided that the time of a single context switch with no required change in address space is determined primarily by the CPU speed, there is a factor of 4.75 in CPU speed (1900 MHz to 400 MHz), compared to a factor of 17.38 (worst case) and 5.17 (best case) in context switch time.

## Measuring Gizmo Exchange

It is crucial for the successful execution of distributed applications in multihop ad-hoc networks, that a sufficiently large number of gizmos can be exchanged during en passant communication. In another set of experiments, this number of transferred gizmos between mobile devices within communication range is determined. The test programs for these experiments are written in C# as mobile applications using the functionality of the GecGo prototype. Varying parameters are the size of state gizmos, the number of participating devices, and the average mobility of specific devices.

The number of gizmos with increasing size transferred successfully between two iPAQs using the first prototype version of GecGo are depicted in table 1. As expected, the number of gizmos transmitted decreases with increasing gizmo size. Obviously, these numbers are still fairly low compared to the available throughput of nominal 11 Mbps offered by the WLAN adapters of the mobile devices. But since

the prototype version of GecGo has not yet been optimized and performance tuned, hopefully there will be space left for improvements. Especially, the frequency of beacon messages, the impact on power consumption, as well as the fine tuning of execution paths during the exchange of gizmos has not been investigated in more detail yet. In order to estimate the possible increase in throughput, it is also intended to measure the achievable number of bytes transmitted over a plain TCP connection in an identical scenario between several iPAQs in an accompanying experiment.

| Size of state gizmo (Bytes) | Transmitted gizmos per second | Throughput (Bytes/s) |
|---|---|---|
| 0 | 11.3 | - |
| 512 | 2.12 | 1083 |
| 1024 | 1.6 | 1638 |
| 2048 | 0.87 | 1774 |
| 4096 | 0.45 | 1843 |
| 8192 | 0.22 | 1775 |
| 16384 | 0.11 | 1757 |

*Table 1: Transmitted gizmos between 2 devices*

Nevertheless, this version of the GecGo prototype can already be used for first experiments with mobile applications: assuming that two devices are within communication distance for at least 50 seconds—provided that two mobile devices with a communication range of 50 meters travel with a speed of about 1 meter per second—they can exchange e.g. about 100 gizmos of size 512 bytes.

The same experimental program can also be used for more than two devices in order to determine the influence of neighboring devices on gizmo exchange. Due to the spread spectrum modulation of wireless communication, the interference between pairs of communicating devices appears to be fairly low. With 4 iPAQs, where gizmos are exchanged among pairs of mobile devices, the number of successfully transmitted gizmos is identical to the 2 iPAQ scenario (2.12 and 2.13 gizmos per second in case of 512 byte size).

## First GecGo Applications

The GecGo middleware platform is currently used to implement several example applications, to gain experience with the abstractions provided by the middleware and to improve the platform architecture and functionality. We started with the development of a simple e-learning application: a peer-to-peer quiz for students to assist in the preparation of examinations. The basic idea is to enable participating students to issue interesting examination questions. These questions are propagated by the GecGo middleware to the corresponding venue that has been assigned to the specific course. Participants interested in examination questions for a given course will issue a request that too will be propagated to the corresponding venue where it remains for some period of time to collect new items. This collection of new questions will be realized by means of additions to the initial mobile state gizmo. Eventually, the request will move back to the sending owner and any results will be presented to the user. Additionally, the application enables students to rate and to order a set of questions from a didactical point of view. Rates and orders are again sent to the venue to be accessible to other participants.

The implementation of additional mobile applications for ad-hoc networks using GecGo is planned for the next 9 months: a mobile auction system and a self-organizing electronic rideboard in an university environment [1,5]. These applications have been investigated already on a simulated basis [3,6] and as prototypes running on a java-based middleware called SELMA [4], the predecessor of GecGo.

## 5. Related Work

Traditional middleware systems such as CORBA, Microsoft DCOM or Java RMI are not suitable for mobile ad-hoc networks because they rely on central infrastructure like naming services and assume the reachability of all network nodes. These assumptions cannot be matched by mobile multihop ad-hoc networks. Additionally, traditional middleware approaches are too heavyweight for mobile devices. Many adaptions have been made to apply them in mobile settings such as OpenCORBA [7] or Next-GenerationMiddleware [8]. These extensions provide mechanisms for context awareness, but cover mainly infrastructure networks and one-hop mobile communications.

An increasing number of middleware systems is developed specifically for mobile ad-hoc networks. XMIDDLE [9] allows the sharing of XML documents between mobile nodes. Lime [10] and L2imbo [11] are based on the idea of tuple-spaces [12], which they share between neighbored nodes. But due to the coupling of nodes, these approaches are not well-suited for highly mobile multihop ad-hoc networks. MESH-Mdl [13] employs the idea of tuple-spaces as well, but avoids coupling of nodes by using mobile agents, which communicate with each other using the local tuple-space of the agent platform. Proem [14] provides a peer-to-peer computing platform for mobile ad-hoc networks. STEAM [15] limits the delivery of events to geographic regions around the sender which is similar to the geographically bound communication at marketplaces. STEAM provides no long distance

communication, it is only possible to receive events over a distance of a few hops.

Mobile agent frameworks exist in numerous variations, Aglets [16] or MARS [17] may serve as examples. These frameworks were designed for fixed networks and thus the above mentioned problems of traditional middleware approaches apply to them as well. The SWAT infrastructure [18] provides a secure platform for mobile agents in mobile ad-hoc networks. This infrastructure requires a permanent link-based routing connection between all hosts and thus limits the ad-hoc network to a few hops and it is therefore not applicable to en passant communication pattern.

## 6. Conclusions

The specific nature of multihop ad-hoc networks enforces a tight coupling between the middleware and any mobile application. The sole dependence on information local to the mobile device leads to new programming and execution models, that favor self-organization and adaption to a continuously changing environment. The specific architecture of the GecGo middleware as presented in this paper is trying to address these issues by supporting mobile application components and by providing flexible interaction mechanisms between entities on a single device as well as entities on mobile devices that are within communication range for short period of times.

One of the major goals of this project is to verify that the system software offered by Microsoft, which addresses wireless infrastructures and mobile applications, also suits application needs in multihop ad-hoc networks. At the time of writing, only preliminary results are available. The successful implementation of the GecGo middleware indicates, that in principle no arguments prohibit the usage of the .NET Compact Framework in such an environment. But in many situations, it was obvious that the current version of the Compact Framework addresses issues in singlehop networks only. In such an environment, the wireless communication facilities are primarily substitutes for a physical wire with the traditional protocol stack on top of it. This is no disadvantage, since it is meant to support exactly this environment. But questions to be answered in the future of this project will address issues on additional support for multihop ad-hoc networks and how to integrate these required functions into existing system software such as the .NET Compact Framework.

## 7. Funding

This work is funded in parts by the german science foundation DFG as part of the Schwerpunktprogramm SPP1140 „Basissoftware für selbstorganisierende Infrastrukturen für vernetzte mobile Systeme“. The Microsoft Windows CE and Microsoft .NET Compact Framework implementation of GecGo is funded by an Microsoft Research Embedded Systems IFP Grant (Contract 2003-210).

## 8. References

[1] H. Frey, J.K. Lehnert, and P. Sturm. "*Ubibay: An auction system for mobile multihop ad-hoc networks*," Workshop on Ad hoc Communications and Collaboration in Ubiquitous Computing Environments (AdHocCCUCE'02), New Orleans, Louisiana, USA, 2002

[2] D. Görgen, H. Frey, J. Lehnert, and P. Sturm, "*Marketplaces as communication patterns in mobile ad-hoc networks*," in Kommunikation in Verteilten Systemen (KiVS), Leipzig, Germany, 2003

[3] J. K. Lehnert, D. Görgen, H. Frey, and P. Sturm. "*A Scalable Workbench for Implementing and Evaluating Distributed Applications in Mobile Ad Hoc Networks*," Western Simulation MultiConference WMC'04, San Diego, California, USA , 2004

[4] D. Görgen, J. K. Lehnert, H. Frey, and P. Sturm. "*SELMA: A Middleware Platform for Self-Organzing Distributed Applications in Mobile Multihop Ad-hoc Networks*;" Western Simulation MultiConference WMC'04, San Diego, California, USA, 2004

[5] H. Frey, D. Görgen, J. K. Lehnert, and P. Sturm. "*Auctions in mobile multihop ad-hoc networks following the marketplace communication pattern*," submitted to 6th International Conference on Enterprise Information Systems ICEIS'04, Porto, Portugal, 2004

[6] H. Frey, D. Görgen, J. K. Lehnert, and P. Sturm. "*A Java-based uniform workbench for simulating and executing distributed mobile applications*," FIDJI 2003 International Workshop on Scientific Engineering of Distributed Java Applications, Luxembourg, Luxembourg, 2003 (to appear in Springer LNCS)

[7] T. Ledoux. "*OpenCorba: A reactive open broker*," Springer LNCS, Volume 1616, pp. 197ff, 1999

[8] G. S. Blair, G. Coulson, P. Robin, and M. Papathomas. "*An architecture for next generation middleware*," in Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing, Springer-Verlag, London, UK, 1998

[9] S. Zachariadis, L. Capra, C. Mascolo, and W. Emmerich. "*XMIDDLE: Information sharing middleware for a mobile environment*," in ACM Proc. Int. Conf. Software Engineering (ICSE02). Demo Presentation, Orlando, Forida, USA, 2002

[10] G. P. Picco, A. L. Murphy, and G.-C. Roman. "*LIME: Linda meets mobility*," in International Conference on Software Engineering, pp. 368-377, 1999

[11] N. Davies, A. Friday, S. P. Wade, and G. S. Blair. "*L2imbo: A distributed systems platform for mobile computing*," ACM Mobile Networks and Applications (MONET) - Special Issue on Protocols and Software Paradigms of Mobile Networks, Volume 3, pp. 143-156, Aug. 1998

[12] S. Ahuja, N. Carriero, and D. Gelernter. "*Linda and friends*," IEEE Computer, Volume 19, pp. 26-34, Aug. 1986.

[13] K. Herrmann,. "*MESHMdl - A Middleware for Self-Organization in Ad hoc Networks*," in Proceedings of the 1st International Workshop on Mobile Distributed Computing (MDC'03), 2003

[14] G. Kortuem. "*Proem: a middleware platform for mobile peer-to-peer computing*," ACM SIGMOBILE Mobile Computing and Communications Review, Volume 6, Number 4, pp. 62-64, 2002

[15] R. Meier and V. Cahill. "*STEAM: Event-based middleware for wireless ad hoc networks*," in 22nd International Conference on Distributed Computing Systems Workshops (ICDCSW '02), Vienna, Austria, 2002

[16] D. Lange and M. Oshima. "*Programming and Deploying Java Mobile Agents with Aglets*," Addison-Wesley, 1998

[17] G. Cabri, L. Leonardi, and F. Zambonelli. "*MARS: A programmable coordination architecture for mobile agents*," IEEE Internet Computing, Volume 4, Numer 4, pp. 26-35, 2000

[18] E. Sultanik, D. Artz, G. Anderson, M. Kam, W. Regli, M. Peysakhov, J. Sevy, N. Belov, N. Morizio, and A. Mroczkowski. "*Secure mobile agents on ad hoc wireless networks*," in The 15th Innovative Applications of Articial Intelligence Conference, American Association for Articial Intelligence, 2003