Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Telecommunications and Telematics

Implementation of a Protocol Stack for Personal Area Networks

Master's Thesis

Kristóf Fodor

Advisors:

Miklós Aurél Rónai

M.Sc., Ericsson Research, Traffic Lab

Zoltán Richárd Turányi

M.Sc., Ericsson Research, Traffic Lab

Róbert Szabó

Ph.D., Budapest University of Technology and Economics

Budapest, 2003.

Nyilatkozat

Alulírott Fodor Kristóf, a Budapesti Műszaki és Gazdaságtudományi Egyetem hallgatója kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, és a diplomatervben csak a megadott forrásokat használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Budapest, 2003. május

....
Fodor Kristóf

Kivonat

Napjainkban az elektronika és a számítástechnika fejlődése következtében egyre több és okosabb készülék vesz minket körül. Az emberek fokozatosan hozzászoknak a létükhöz, majd kis idő elteltével már nélkülözhetetlennek tekintik ezeket az eszközöket, sőt további igényeket fogalmaznak meg velük szemben. Ilyen igényre példa azon elvárás, miszerint az intelligens készülékek folyamatosan figyeljék a környezetükben levő személyek szándékait és segítsenek nekik ezek végrehajtásában.

Mark Weiser 1991-ben fogalmazta meg elképzelését egy olyan világról, melyben az intelligens eszközök láthatatlanul beépülnek mindennapjainkba, azok szerves részévé válnak, és úgy segítenek nekünk felhasználóknak, hogy az fel sem tűnik. Eme elképzelés azon a gondolaton alapszik, hogy a készülékek alkalmazkodnak a felhasználóhoz, és nem a felhasználónak kell megtanulnia kezelnie a különböző eszközöket. Jelenleg a világban számos kutatás folyik e területen, amit összefoglalva *ubiquitous computing*nak vagy más néven *pervasive computing*nak hívnak.

Diplomamunkám első részében röviden bemutatom a ubiqitous computing tudományterületet és főbb kihívásait, majd vázolom eme kutatási területen belül a világban jelenleg futó jelentősebb projekteket és áttekintem a megvalósításaikkal kapcsolatos irodalmat.

A diplomamunka második részében ismertetem a Blown-up elképzelést, mely a ubiqitous computing területen belül az eszközök, azon belül az általuk nyújtott szolgáltatások összekapcsolását célozza meg. Az elképzelés leírása után részletesen bemutatom a Blown-up Micronet Protocol-t, amely segítségével egy felhasználó környezetében levő szoftver és hardver elemek egyesíthetőek egy feladat végrehajtásának céljából. A Blown-up rendszer például lehetőséget ad arra, hogy a kényelmesebb használat kedvéért egy digitális személyi asszisztenshez (Personal Digital Assistant – PDA) hozzákapcsoljunk egy normál méretű billentyűzetet, illetve egeret. Elképezelhető az is, hogy egy PDA-n futó zenelejátszó programot összekössük egy hangszoróval, ezáltal a zenehallgatásra a jobb minőségű hangot adó külső eszközt használjuk. A protokoll elfedi a szolgáltatások elől azok elosztottságát: mindegyik szolgáltatás úgy látja a másikat, mintha azokat ugyanazon eszköz nyújtaná. A fejezet lezárásaként ismertetem a protokollhoz tartozó programozói interfészt.

Diplomamunkám harmadik részében bemutatom az általam megvalósított Blown-up rendszert és programozói interfészét, továbbá a környezetet, amiben a rendszert kifejlesztettem. Kitérek a belső felépítésére, majd az osztályok főbb függvényeire és főbb struktúráira. Továbbá ismertetem a Blown-up rendszerek egymás közötti, illetve a Blown-up rendszerek és programok közötti kommunikáció folyamatát. Végezetül vázolom programom tesztelésének menetét, és az implementáció során szerzett tapasztalatok alapján javaslatokat teszek a protokoll módosítására.

Abstract

Nowadays—in consequence of the progress of electronics and computer technology—more and more intelligent devices surround us. People continuously get accustomed to their presence, and after a while people will consider these devices as an indispensable part of their life, moreover they will conceive new claims. An example for this expectation is the claim for intelligent devices that continuously follow the intentions of the users with attention, and help with their execution.

In 1991 Mark Weiser drafted his concept about a world, wherein devices invisibly integrate into our everyday lives, they become a natural part of it, and help us—users—seamlessly. This concept is based on the idea that devices adapt themselves to the users, and not the user is the one to learn the usage of them. At present, there are many research projects in this field that is called *ubiquitous computing* or *pervasive computing* in general.

In the first part of my thesis I present the research field of ubiquitous computing with its challenges, then I introduce the ongoing projects in this area. Moreover I discuss the literature about the realisation of these ongoing projects.

The second part of my thesis contains the presentation of the Blown-up concept that aims connecting applications offered by devices within the field of ubiquitous computing. After the introduction of the concept I discuss the Blown-up Micronet Protocol in details. This protocol is designed to unite software applications and hardware devices (commonly called services), located in the near of a user, to execute a task of the user. For example the Blown-up system provides facility to connect a normally-sized keyboard, mouse and display to a Personal Digital Assistant (PDA) enabling its owner to use the PDA more comfortable. It is also conceivable to connect a music player application on a PDA to a speaker, hereby using a speaker of better quality to listen to music. The protocol masks the distributedness of the services: each service appears to the other services as they were located on the same device. At the end of the chapter I present the application programming interface defined for the protocol.

In the third part of my thesis I discuss the Blown-up system that I have created, moreover its application programming interface and the environment wherein I have implemented the system. Then I present its architecture and also its main functions and registries. Furthermore, I describe how the Blown-up systems communicate with each other, and how local services can connect to them. Finally, I discuss the testing of the system, and I give recommendations to the modifications of the protocol based on experiences of the implementation.

Acknowledgments

I would like to thank Gergely Biczók, Balázs Kovács and Ágoston Szabó for their support with whom I have worked together in the last two years.

Special thanks are also due to Miklós Aurél Rónai, alias Hawai who stood always by me during my stay at the Ericsson Traffic Laboratory. Furthermore I would like to give thank to Zoltán Richárd Turányi and András Gergely Valkó for their useful advices.

Contents

Abbreviations

ACL – Access Control List

ACK – Acknowledgement

AODV – Ad hoc On-demand Distance Vector

API – Application Programming Interface

BUMP – Blown-Up Micronet Protocol

BUMPC – BUMP controller

CA – Control Application

CFS – Cooperative File System

CommC – Class Communication

DSDV – Destination-Sequenced Distance Vector

DSR – Dynamic Source Routing

E21 – Enviro21

GUI – Graphical User Interface

H21 – Handy21

IC – Integrated Circuit

IEEE – Institute of Electrical and Electronics Engineers

IETF – The Internet Engineering Task Force

INS – Intentional Naming System

IP – Internet Protocol

N21 – Network21

NL – Network Layer

O21 – Oxygen software

PAN – Personal Area Network

PDA – Personal Digital Assistant

REP	Remote Event Passing
RON	Resiliant Overlay Network (RON)
SFS	Self-Certifying File System
TAP	Transport Access Point
TCP	Transmission Control Protocol
TL	Transport Layer
TORA	Temporarily Ordered Routing Algorithm
UA	User Applications

List of Figures

Chapter 1

Introduction

”What information consumes is rather obvious: it consumes the attention of its recipients. Hence a wealth of information creates a poverty of attention, and a need to allocate that attention efficiently among the overabundance of information sources that might consume it.”

Herbert Simon (1971)

Nowadays a claim conceives to support people to access their personal data anytime and anywhere. At an early stage the disks where an ideal solution for this purpose: data could be transported from a computer to another. With the evolution of computer networks, the accessing of personal data becomes more easier. At present the laptops and Personal Digital Assistants (PDA) are more and more spreading, which enables to the people to have their own data, furthermore their personal computing environment always accessible. However the usage of PDAs could be uncomfortable, because of the PDA's small size.

There is a need for solutions that enables the linking of normally-sized, easily connectable devices to a PDA, which could extend the usability of the PDA's built-in peripherals. Let us assume that somebody uses his PDA on the way to his office, on a train or on a bus, and when he arrives in his office then he connects his PDA to the local wireless keyboard, mouse and display, and keeps on working on the PDA, but more comfortable from now on. Furthermore in the future it is conceivable to travel on trains and buses that have displays, keyboards and pointer devices built in the seats, which can be connected to the PDAs of the users.

Generalizing the previously presented problem, by considering displays, keyboards and pointer devices as services, a concept can be outlined wherein services can be connected to PDAs. Moreover—improving the concept and considering to the programs on the PDAs also as services—services can be connected to other services. Thus, different services located in an ad hoc network can be joined to execute a complex user task. If this ad hoc network consist of devices that are located near by the user then it can be called *Personal Area Network* (PAN).

In my thesis I present an idea called Blown-up concept that is based on the previously drafted concept, furthermore I show a realisation of this system, which I have created. The name Blown-up comes from the feature of the concept: it connects distributed software and hardware devices (common called services) over a basically wireless Personal Area Network. In the PAN all the services perceive the other services as they were offered by the same device, i.e. the concept Blown-up unites devices into one virtual device. Hence application programmers do not need to know the actual structure of the ad hoc network, they can implement their own programs as a collection of connected local services.

The Blown-up concept can be best classified to the field of ubiquitous and pervasing computing. The goal of these research areas is to create intelligent environments that perceive the intentions of the users and based on this knowledge they automatically support the tasks of the users. Blown-up is a solution that can seamlessly connect the devices in an intelligent environment.

Structure of the Thesis

In my thesis I discuss the Blown-up concept, furthermore the implementation of the system that I have implemented to realise the concept. In Chapter 2 I describe the research areas ubiquitous computing and pervasive computing with the challenges posed to this new field. Furthermore I present four projects and their realisation in the field of ubiquitous and pervasive computing.

In Chapter 3 I introduce the Blown concept and the Blown-up Micronet Protocol (BUMP). I discuss the protocol stack and present the functions of the layer. Furthermore I describe the messages used in the protocol and the functions of the Application Programming Interface that were defined to help developers by creating applications over the BUMP.

In Chapter 4 I present the Blown-up system that I have implemented. I introduce the classes the implementation consist of, and in addition I discuss the main registries and functions of the classes. I introduce the testing method of the system and I evaluate the BUMP. Furthermore I present some features that could be added to the protocol in the future.

In Chapter 5 I summarize the thesis.

Chapter 2

Related work

2.1 Marks Weiser's vision

"The most profound technologies are those that disappear.
They weave themselves into the fabric of everyday life
until they are indistinguishable from it."

So began Mark Weiser's 1991 paper [Weiser91] that first drafted the principles of ubiquitous computing. As the leading developer of Xerox, Mark Weiser created various prototypes of products based on his unique ideas. He states in his article that information technology should become a natural part of people's everyday lives, usage of devices should be just as evident as, for example, reading. When you read a sign on the street you absorb its information without consciously performing the act of reading. In his opinion the most efficient technologies are the ones that the people, whilst actually using it, are not aware of the usage.

His conception is the opposite of the paradigm of virtual reality, since the latter focuses an enormous apparatus on simulating the world rather than on invisibly enhancing the world that already exists. In his opinion ubiquitous computing should explore quite different ground from the idea that computers should be autonomous agents that take on our goals [Weiser93]. To characterize the difference he describes an example. Suppose you want to lift a heavy object. You can call in your strong assistant to lift it for you, or you can be yourself made effortlessly, unconsciously, stronger and just lift it. There are times when both are good. Much of the past and current effort for better computers has been aimed at the former; ubiquitous computing aims at the latter.

To understand the point of ubiquitous computing Mark Weiser described an example with a girl called Sal, some devices in the background and the intelligent softwares running on these devices.

Sal awakens: she smells coffee. A few minutes ago her alarm clock, alerted by her restless rolling before waking, had quietly asked "coffee?", and she had mumbled "yes". "Yes" and "no" are the only words it knows. The alarm clock sends her request to the coffee machine.

At breakfast Sal reads the news. She still prefers the paper form, as do most people. She spots an interesting quote from a columnist in the business section. She wipes her pen over the name of the newspaper, date, section, and page number and then circles the quote. The pen sends the quote to her office.

Electronic mail arrives from the company that made her garage door opener. She lost the instruction manual, and asked them for help. They have sent her a new manual, and also something unexpected – a way to find the old one. According to the note, she can press a code into the opener and the missing manual will find itself. In the garage, she tracks a beeping noise to where the oil-stained manual had fallen behind some boxes. Sure enough, there is the tiny device the manufacturer had affixed in the cover to try to avoid e-mail requests like her own.

Once Sal arrives at work, the foreview helps her to quickly find a parking spot. As she walks into the building the machines in her office prepare to log her in, but don't complete the sequence until she actually enters her office. The telltale by the door that Sal programmed her first day on the job is blinking: fresh coffee. She heads for the coffee machine.

Coming back to her office, Sal picks up a device and "waves" it to her friend Joe in the design group, with whom she is sharing a virtual office for a few weeks. They have a joint assignment on her latest project. Virtual office sharing can take many forms – in this case the two have given each other access to their location detectors and to each other's screen contents and location. Sal chooses to keep miniature versions of all Joe's shared devices in view and 3-dimensionally correct in a little suite of tabs in the back corner of her desk.

She can't see what anything says, but she feels more in touch with his work when noticing the displays change out of the corner of her eye, and she can easily enlarge anything if necessary.

A device on Sal's desk beeps, and displays the word "Joe" on it. She picks it up and gestures with it towards her liveboard. Joe's face appears on the liveboard and they start talking. Joe mentions a colleague called Mary but Sal cannot recall her face. She only remembers that they met on a meeting about a week ago. Sal starts a quick search for the photo and biography of Mary's among the people who participated the same last week as Sal.

This example above shows a world visualized more than ten years ago. In view of the realizations today the visions are nowadays of course different. However, the example above can be a good starting point to understand the concept for the ones who hear about ubiquitous computing for the first time.

2.2 Expectations and challenges

At the examination of the challenges posed by pervasive computing—that is a synonym for ubiquitous computing[1]—I take M. Satyanarayanan's article written in 2001 as a basis [Satya01]. He outlines a progression from distributed computing to mobile computing and to pervasive computing. He observes, that key issues in distributed computing include

- *remote communication*, including protocol layering, remote procedure call, the use of timeouts, and the use of end-to-end arguments in placement of functionality
- *fault tolerance*, including atomic transactions, distributed and nested transactions, and two-phase commit
- *high availability*, including optimistic and pessimistic replica control, mirrored execution, and optimistic recovery
- *remote information access*, including caching, function shipping, distributed file systems, and distributed databases

[1]a study presented at the end of this section will show the relation between the two notions

- *security*, including encryption-based mutual authentication and privacy.

Satyanarayanan states: issues in mobile computing are a cross product of the above issues with new issues such as mobile networking, mobile information access, adaptive applications, energy awareness, and location sensibility. In his article he finally considers four main aspects to create a technology that can invisibly assimilate into our everyday lives.

The first aspect is the *usage and integration of smart spaces.* Smart spaces are intelligent computer systems installed in common buildings, rooms, etc. When used efficiently, smart spaces are e.g. able to control the buildings features like heating and lighting of rooms according to the people's position and actions. In another point of view, in smart spaces a software used by a person can change behavior according to the user's position.

The second aspect is *invisibility* – according to the vision of Mark Weiser pervasive computing has got to exclude consciousness from the operation. In practice, a reasonable approximation to this ideal is minimized user distraction. If a pervasive computing environment continuously meets user expectations and rarely presents him or her with surprises it allows interaction nearly on subconscious level.

The third is *local scalability* – as the size of a smart space grows the number of participating devices and hence the number of interactions between the user and the surrounding entities increase. This can lead to lack of bandwidth, more power consumption and hence inconvenience for the users. The presence of multiple users will further complicate the problem. Previous works on scalability ignored physical distance – a web server should handle as many clients as possible regardless of whether they are located next door or across the country. In pervasive computing the number of interactions should decrease if the distance between the user and the smart space increases otherwise the system will be overwhelmed with interactions of little relevance. It is also important to allow users to send requests to a smart space from thousands of kilometers away.

The last one is the *ability of masking areas with uneven conditions.* The penetration of ubiquitous computing is dependent of many non-technical factors like organizational structure, economics and business models. Uniform penetration, if ever achieved, is many years or decades away. Hence the difference between the ”smartness” of different areas will be huge. There surely will be offices and buildings with more modern equipment than others. These differences can be jarring to a user, which contradicts the goal of creating an invisible computing infrastructure. One way to reduce the amount of variation seen by a user is to have his or her personal computing space for ”dumb” environments. As a trivial example, a system that is capable of disconnected operation able to mask the absence of wireless coverage in its environment.

In the recent decade, a number of scientific achievements were made which are of great importance in connection with pervasive computing and materializing Weiser’s idea. These include the use of fiber optics in data transmission [Gilder93] (that provides almost limitless bandwidth), the evolution of human voice controlled systems [Tatai97, Multivox92]—which is needed for new generation user interfaces—and the breakthrough in image processing [Roska&Chua93]. Nowadays researches based on wireless ad hoc networks are of great interest. Among others, during these researches new technologies are discovered that can be used in the realization of ubiquitous computing. Various radio interface technologies were developed for supporting communication: IEEE 802.11 [WLAN99], HiperLAN and HiperLAN2 [HLAN2] and Bluetooth [JH98, BBSpec]. There are also various routing algorithms for ad hoc networks like AODV (Ad hoc On-demand Distance Vector), DSR (Dynamic Source Routing) [DPR00], DSDV (Destination Sequenced Distance Vector) [PB94] and TORA (Temporarily Ordered Routing Algorithm) [PC97, BMJHJ98].

Despite the achievements mentioned above, there are several fields of research waiting for new results. The main arising problems of development and realization of a pervasive system are nowadays:

- tracking user intentions
- exploiting wired infrastructure to relieve mobile devices
- adaptation strategies: applications must adapt to the needs of the system and the system must be able to adapt to the needs of the applications as well (QoS)
- high level energy management, physical and performance planning
- context awareness
- equilibrium between proactivity and invisibility

- security and authentication
- merging of pieces of information from different levels (it might be useful to extend a low-level resource information with a higher level context information)

In his article Mark Weiser named the new field of computer science based on his idea ubiquitous computing. However, recent documents refer to the same subject as pervasive computing which naming stems from researchers of IBM. The connection between these two expressions is often mentioned and there is also a whole article dealing with the matter [McCrory00]. Some consider the two expressions as synonyms, pervasive computing is simply a new name of ubiquitous computing. Others think that the two expressions mean quite the same with some differences: pervasive computing is based on a system small and mobile devices that is used for retrieving information anytime, anywhere (e.g. surfing in the Internet using a cellular phone), while the goal of ubiquitous computing is to hide computer architecture. In recent documents there is also an other name for this field of computer science: invisible computing. In the thesis I consider these expressions as synonyms of each other.

In the following sections of this chapter I will present some ongoing projects in the field of pervasive computing emphasizing—as far as it is published—their architecture and connection management.

2.3 AURA

Today, a major source of user distraction arises from the need for users to manage their computing resources in each new environment, and from the fact that the resources in a particular environment may change dynamically and frequently. In Project Aura [Aura02a, Aura02b] at Carnegie Mellon University the researchers are developing a new solution to this problem based on the concept of personal Aura. The intuition behind a personal Aura is that it acts as a proxy for the mobile user it represents: when a user enters a new environment, their Aura marshals the appropriate resources to support the user's task. Furthermore, an Aura captures constraints that the physical context around the user imposes on tasks.

To enable the action of such personal Aura, an architectural framework is needed that clarifies which new features and interfaces are required at system- and application-level. The framework must have also defined placeholders for capturing the nature of the user's tasks, personal preferences, and intentions. This knowledge is key to configure and monitor the environment, thus shielding the user from the heterogeneity of computing environments as well as from the variability of resources.

Figure 2.1 shows a bird's-eye view of the proposed architectural framework by the researcher at the Aura project. There are four component types: first, the *Task Manager*, called *Prism*, embodies the concept of personal Aura. Second, the *Context Observer* provides information on the physical context and reports relevant events in the physical context back to Prism and the *Environment Manager*. Third, the Environment Manager embodies the gateway to the environment; and fourth, *Suppliers* provide the abstract services that tasks are composed of: text editing, video playing, etc.

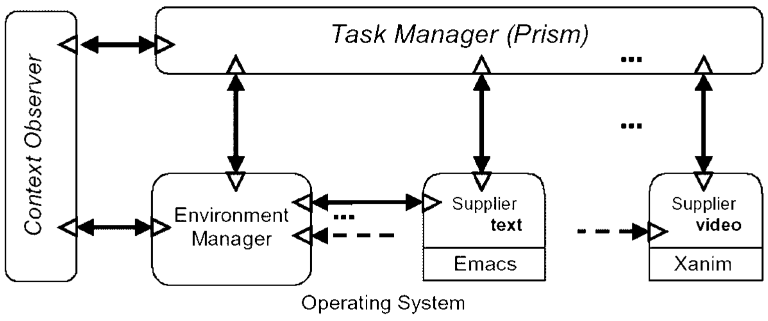


Figure 2.1: Aura bird's-eye view [Aura02b]

The environments are not defined by the physical boundaries of boxes or by network connectivity, they are only of administrative nature. For simplicity, the developers of the framework consider that each environment has one running instance of each of the types: Environment Manager, Context Observer and Task Manager. Naturally, components of these types cooperate with the corresponding components in other environments. One environment has several service Suppliers: the more it has, the richer the environment is.

The Task Manager (Prism) has to handle the changes of the environment and the context. It must minimize the user distractions in four fields:

- *user migration between two environments*: if user moves into a new environment Task Manager has to coordinate the migration all the data the user task uses and negotiate the task support with the new Environment Manager.
- *changes in environments*: if Quality of Service informations provided by components becomes incompatible with the requirements of the current task, Prism has to query the Environment Manager to find a new configuration to support the task. The same operation has to be done if a monitored component dies.
- *changes in tasks*: if Prism notices that the user interrupts his current task or switches to a new task, the Task Manager coordinates saving the state of the interrupted task and instantiates the intend new task. To find out the user's intentions Prism monitors explicit indications from the user and receives event announcements from the Context Observer.
- *changes in contexts*: when constraints on the context included in the task description are not met, Prism restricts allowed operations. The researcher at the Carnegie Mellon University gives an example: when a user works with sensitive data and a second person whom is not allowed to see the informations enters the room then the display will be automatically hidden by the Task Manager.

The key idea behind Prism is a platform-independent description of user tasks [Aura00]. The earlier research in this area treated task as a cohesive collection of applications, this was extended at the project Aura by describing task as a coalition of abstract services, such as ”edit text” and ”play video”. In this way tasks can be successfully instantiated in different environments using different supporting applications.

Context Observers provide information on the physical context and report relevant events in the physical context back to Prism and the Environment Manager. It can have different degree of complexity in each environment, depending on the number and capabilities of the sensors located in the environment. In case of a powerful Context Observer Prism has less job, it is less occupied with detecting and finding out user intentions from explicit indications.

The Environment Manager is the gateway to the environment: it is aware of which components are available to supply which services, and where they can be employed. It also encapsulates the mechanism for distributed file access. Every Suppliers installed in an environment is registered with the local Environment Manager, so when a new task is initiating and needs an abstract service, the discovery mechanisms only have to look at this registry.

Suppliers provide the abstract services that tasks are composed of. Two suppliers for the same type of service can be different depending how complex they are. For instance a supplier for text editing can be more powerful than an other, if in addition to the compared one it has a spell checking function built in. For task migration purposes, service suppliers have to possess a function to extract and map service status information to and from the Task Manager and also use an universal representation of service status. For the latter one a markup representation is used which contains a vocabulary of tags and the corresponding interpretation. Each service type is characterized by a distinct vocabulary of tags corresponding to the information relevant for the service, although there are commonalities across service types.

All the component types in Aura's architecture have standard interfaces, or ports (represented by triangles in Figure 2.1). These ports only support local method calls. When Prism migrates a task from one environment to another, the deployment of the suppliers across devices may be very different. To enable dynamic reconfiguration in a transparent way to the involved components and to hide the variation of low-level interaction mechanisms from one environment to the next, Aura uses a feature called *connector*. Connectors are autonomous pieces of code that are assuring the interconnection between a local and a remote port. There are four types of them in the Aura architectural framework: one type between Prism and an arbitrary supplier, one type between Prism and the Environment Manager, and two other types connecting the Context Observer to Prism and to the Environment Manager. Each of these connector types is defined by an interaction protocol appropriate to the component type it connects.

The developers at Carnegie Mellon University have built an experimental system based on the presented architecture. As it can be seen on the video at the homepage of Aura, the campus-wide system works quite well. They have lot of useful applications over their architecture that allow us to imagine a way of life in the era of ubiquitous computing.

2.4 Oxygen

The Project *Oxygen* [Oxygen] running at Massachusetts Institute of Technology (MIT) enables pervasive, human-centered computing through a combination of specific user and system technologies. They directly address human needs using speech and vision technologies that enable the user to communicate with Oxygen as if he were interacting with another person. Automation, individualized knowledge access, and collaboration technologies help users to live in a comfortable world, where they can do lot of things easier or at all.

The devices, networks and applications used by Oxygen (Figure 2.2) extends the range of a user by delivering the technologies to the user's home, at work or on the go. The researcher have designed two type of devices: the so called *Enviro21* computational devices (*E21s*) placed in homes, offices and cars to sense and affect the users immediate environment, and the so called *Handy21* handheld devices (*H21s*) to provide communication and computing support to the users no matter where they are. Both mobile and stationary devices are universal communication and computation appliances. They are also anonymous: they do not store configurations that are customized to any particular user. The primary difference between them lies in the amount of energy they supply. The researcher also invented special softwares (*O2s*) running on E21 and H21 devices which are able to adapt to changes in the environment or in user requirements. To locate devices, access resources, people or services dynamic, self-configuring networks, so called *N21s* were designed.

Collections of the E21 embedded devices create *intelligent spaces* inside offices, buildings, homes, and vehicles (Figure 2.2). E21s provide large amounts of embedded computation, as well as interfaces to camera and microphone arrays, large area displays, and other devices. Users communicate naturally in the spaces created by the E21s, using speech and vision, without being aware of any particular point of interaction.

H21 handheld devices provide mobile access points for users both within and without the intelligent spaces controlled by E21s. H21s accept speech and visual input, and they can reconfigure themselves to support multiple communication protocols or to perform a wide variety of useful functions (e.g., to serve as cellular phones, beepers, radios, televisions, geographical positioning systems, cameras, or personal digital assistants). H21s can conserve

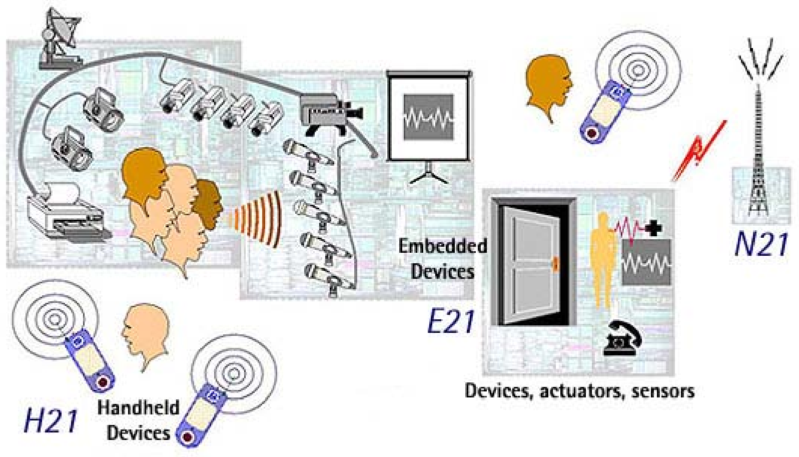


Figure 2.2: The components of Oxygen [Oxygen]

power by offloading communication and computation onto nearby E21s.

The O2 softwares used by Oxygen were built considering the frequent changes of the environment. There can be many reasons for alterations, for example it can be occasioned by anonymous devices customizing to users, by explicit user requests, by the needs of applications and their components, by current operating conditions, by the availability of new software and upgrades or by failures. Oxygen's software architecture relies on control and planning abstractions that provide mechanisms for change, on specifications that support putting these mechanisms to use, and on persistent object stores with transactional semantics to provide operational support for change.

N21 networks connect dynamically changing configurations of self-identifying mobile and stationary devices. They integrate different wireless, terrestrial, and satellite networks into one global seamless network. Through algorithms, protocols (be presented later), and middleware, N21 networks realize four purposes.

First, N21s *automatically configure collaborative regions*, in which are dynamically self-organizing collections of computers that share some degree of trust. In addition N21s create topologies and adapt them to mobility and change.

Second, the networks used by Oxygen *provide automatic resource and location discovery* enabling the applications to use *intentional names* and location discovery through proximity to named physical objects (e.g. transmitting radio frequency beacons). With intentional names not only statically named resources can be found, but entities too that are characterized by their feature or functionality. The researcher of Oxygen describe some example, for instance using this solution a full soda machine or a surveillance camera that have recently detected suspicious activity can easily be found.

Third, N21 networks *provide secure, authenticated and private access to networked resources.* The base of the security is the collaborative region, in which the devices were instructed by their owners to trust each other to a specified degree. Rules are defined, which are specifying what is allowed and forbidden, for instance in the collaborative region of a meeting guest's are not allowed to use the local printer. Resource and location discovery systems address privacy issues by giving resources and users control over how much to reveal.

Fourth, N21s *adapt to changing network conditions*, including congestion, wireless errors, latency variations, and heterogeneous traffic, by balancing bandwidth, latency, energy consumption, and application requirements. They allow devices to use multiple communication protocols, as well vertical handoffs among these different protocols. N21s provide interfaces to monitoring and control mechanisms, which enable applications to use their own settings by a connection and vary them upon the current situation.

N21 networks use two types of routing protocols: intra-space and wide-area. Intra-space routing protocols perform resolution and forwarding based on queries that express the characteristics of the desired data or resources in a collaborative region. Wide-area routing uses a scalable resolver architecture; techniques for soft state and caching provide scalability and fault tolerance. There are also two types of name resolution solutions: early and late binding between names and addresses (i.e., at delivery time). The preceding supports high bandwidth streams and anycast, the latter one mobility and multicast.

Oxygen integrates many existing protocols and solutions. For routings it use a routing protocol called *Grid*. It was designed for ad-hoc mobile networks, is self-configuring, requires none fixed infrastructure, is robust in the face of node failures and intrinsically supports mobile hosts. For the maintenance of an energy efficient, ad hoc wireless network's topology Oxygen has chosen the *Span* protocol. Span nodes save power by turning off their radio receivers most of the time. A *Resiliant Overlay Network* (*RON*) allows distributed Internet applications over the network to detect and recover from path outages and periods of degraded performance within several seconds. RON nodes monitors the functioning and quality of the Internet paths among themselves, and route packets according to this informations. *Cord*, a scalable distributed lookup protocol fo peer-to-peer networks is also integrated in Oxygen. It maps keys to nodes, adapting efficiently as nodes join and leave the system. The *Cooperative File System* (*CFS*) is based on Cord and provides highly available, read-only storage to a group of cooperating users.

For resource discovery purposes Oxygen uses the *Intentional Naming System* (*INS*). It supports scalable, dynamic resource discovery and message delivery. As mentioned above INS describe application intent in the form of properties and attributes. The resource looked for by a user or service can either be public or protected by SPKI/SDSI (Simple Public Key Infrastructure / Simple Distributed Security Infrastructure) *access control lists* (ACLs). In case of protected resource, software proxies (*K21s*) for resources and users presents authorization information as an answer for an intentional query requested by an INS. The INS then compares this information to resource-supplied ACLs, and if user has access to the requested resource, INS adds it to the list of available requested resources.

Managing session-specific application state across changes in network attachment points and during periods of disconnectivity Oxygen uses a solution, called *Migrate*. The *Self-Certifying File System* (*SFS*), a secure decentralized global file system, is needed to enable users access their data from any location.

2.5 one.world

The researcher at University of Washington, Seattle argue that existing operating system abstractions and services are neither sufficient nor necessarily appropriate for a pervasive computing infrastructure. They claim it is necessary to define a fundamentally new computing platform or system architecture that runs across all devices. The new architecture should:

- *be very simple* to make it possible implementing the architecture across the range of devices

- *integrate the base function needed for mobile computation and persistent storage* to support applications working reliably under limited connectivity
- *be easy to encapsulate*, so that applications can be effectively secured and managed while also preserving local autonomy.
- *expose change*, including failures, rather than hide distribution in order to enable applications implementing their own strategies for handling changes
- *compose dynamically*, so that applications and services can be easy composed and extended by the system at runtime
- *separate data and functionality* enabling their distinct management to evolve independently.

The system architecture called *one.world* [one.world00, one.world01, one.world03], designed at the Department of Computer Science and Engineering as a part of Project Portolano [PortoWeb99, Porto99], directly addresses the concerns of heterogeneous devices, dynamically changing system, limited connectivity, and autonomous administration in order to make it feasible to design, develop, and deploy applications on a global platform. Each devices in the architecture runs a single instance of one.world, which is administrated separately by the owner of the node on which it is running. Applications run within one.world and all of them running on the same node share the same instance of the architecture. The architecture provides the same basic abstractions and core services across all nodes and uses mobile code to provide a uniform and safe execution platform.

The architecture of one.world relies on three main abstractions:

- *components* represent computations; they execute code, possibly, using multiple threads, and have their own state. They statically declare which event handlers they import and export but are dynamically linked and unlinked.
- *tuples* represent persistent data in form of records with named fields. The records are self-describing in that an application can dynamically determine a tuple's fields and their types. The tuples are composed from components.

- *environments* provide structure and control. They integrate tuples, components and other embedded environments. Each application has at least one environment, but at the same time it may span several, nested environments. Applications are informed of important events by their environments and can interpose on their interactions with the kernel and the outside world. one.world's kernel is hosted by each node's root environment.

Figure 2.3 shows an example environment hierarchy. The root environment "I" hosts the kernel of one.world and has one child, called "replicator", which also contains active components. The replicator environment has in addition two children, named "log" and "application". The former child only stores tuples, however the application environment also contains active components.

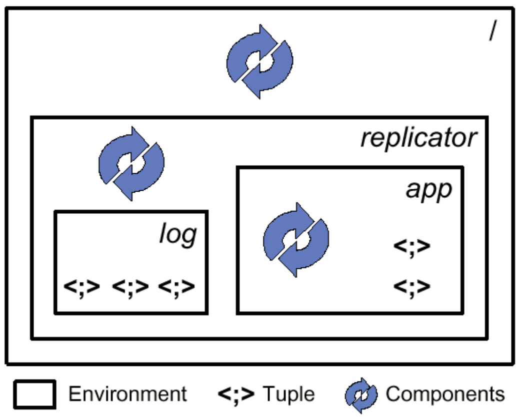


Figure 2.3: An example environment hierarchy [one.world03]

In addition to the basic abstractions, one.world provides a set of services that serve as common building blocks and help developers in making their applications adaptable. *Operations* help manage asynchronous interactions. They simplify interactions that may fail (e.g. sending an electronic mail) by keeping the state associated with event exchanges and by providing automatic timeouts and retries. *Migration* provides the ability to move or copy an environment and its contents, including tuples, components, and nested environments, either locally or to another node. It is especially useful for applications that follow a user from shared device to shared device as he or she moves through the physical world. *Checkpointing* captures the execution state of an environment tree and saves it as a tuple, making it possible to later revert the environment tree's execution state. Checkpointing simplifies the task of gracefully resuming an application after it has been dormant or after a failure, such as devices' batteries running out. *Remote Event Passing* (REP) provides the ability to send events to remote services and is one.world's basic mechanism for communication across the network. To use REP, services export event handlers under symbolic descriptors, that is, tuples, and clients send events by specifying the symbolic receiver. Finally, *discovery* routes events to services with unknown location. It is especially useful for applications that migrate or run on mobile devices and need to discover local resources, such as a wall display or printer.

Data Management

As described above, data management in one.world is based on tuples. They define a common data model, including the type system, for all applications and thus make it easy to store and exchange data. Tuples are self-describing, mutable records with named and optionally typed fields and can be nested within each other. They all have a globally unique identifier to support symbolic references and a metadata field to support applications-specific annotations.

Access to both local and remote resources is controlled by *leases*. Leases limit the time applications can access resources, such as an environment's tuple storage or a communication channel, and force applications to periodically renew their interest in the resources. As a result, leases make time visible throughout the system and cleanly expose change to applications. When an application wants to store some data or communicate with an other application, it *binds* locally to a tuple storage or remote to a communication end-point over TCP or UDP. Then it performs so called *structured I/O operations* that can be:

- *put*: write the tuple
- *read*: read a single tuple
- *listen*: read several tuples as they are written
- *query*: query for several tuples from storage
- *delete*: delete a tuple from storage.

The five presented operation compose a common interface to storage and communications. Operations are atomic, but in case of storage they can optionally use transactions to group several operations into one atomic unit.

The researcher at the project have chosen tuples instead of byte strings for I/O because tuples preserve the structure of data. In addition that tuples obviate the need for explicit marshaling and unmarshalling of data and enable system-level query processing, the use of tuples makes easier to share data between multiple writers. XML is also a possibility to describe and handle data, but the researcher believe its syntax and structure is more complicated than tuple's and its interfaces to access XML-based data (e.g. DCOM) are more complex too, so they have rejected this solution for their architecture.

Events and Components

Control flow in one.world is expressed through asynchronous events that are processed by event handlers. Events are simply tuples, however in addition to the common tuples they have a source field referencing an event handler. This event handler receives the response for a query and the notification of exceptional conditions during event delivery and processing. Furthermore all events have a closure field that is returned with the response in case of a request/response interaction. The use of closure fields simplifies the implementation of event handlers enabling tuples to store additional state needed for processing responses. Event handlers implement a uniform interface with a single method that takes the event to be processed as its only argument.

Application functionality is implemented by components, which import and export asynchronous event handlers exposing them for linking and are instantiated within specific environment. Although imported and exported event handlers can be added and removed after component creation, they are typically declared in a component's constructor. Components can be linked and unlinked at any time. By starting an application, his main component's static initialization method instantiates the application's components and performs the initial linking. While application is running, it can instantiate additional components, add and remove imported and exported event handlers, and relink and unlink components as needed. Event delivery has ”at-most-one”[2] semantics, both for local and remote event handling.

To support asynchronous event handling, each environment provides a queue of pending <event handler, event> invocations as well as a pool of one or more threads to perform such invocations. When sending an event between components in different environments, the corresponding event handler invocation is automatically enqueued in the <event handler, event> queue of the target. When sending an event between components in the same environment, the event handler invocation is implemented as a direct method call.

As mentioned before, *Remote Event Passing* provides the ability to send events to remote receivers. It supports both point-to-point communications and service discovery through three simple operations:

- *export* operation makes an event handler accessible from remote nodes through a symbolic descriptor, that is, a tuple by automatically forwarding its exported <event handler, descriptor> binding to the current discovery server
- *send* operation sends an event to previously exported event handlers by using an exported descriptor or discovery query as the remote address
- *resolve* operation looks up event handlers on the discovery server which keeps <event handler, descriptor> bindings of.

The researcher at University of Washington chose asynchronous events instead of synchronous for three reasons. First, asynchronous events provide a natural fit for pervasing computing, as applications often need to raise or react to events. Second, unlike threads, which implicitly store execution state in registers and on stacks, events make the execution state explicit. Third, the researcher believed that asynchronous events scale better across different classes of devices than threads. A uniform event handling was chosen because it greatly simplifies composition and interposition.

[2]”at-most-one” means that events are not delivered more than once

Environments

As described in the introduction of the architecture one.world, environments integrate tuples, components and other environments. They provide structure and control by grouping data and functionality, and they provide control by nesting environments within each other. The first one, grouping data and functionality, is relevant for checkpointing, and migration when the execution state of an environment tree needs to be captured and restored. When capturing execution state, one.world first quiesces all environments, that is, it waits for all threads to return to their thread polls. It then serializes the affected application state, that is all components in the environment tree, and the corresponding environment state, the <event handler, event> queues. When restoring execution state, one.world first deserializes all applications' and environments' state, then reactivates all threads, and finally notifies applications that they have been restored or migrated.

The nesting of environments is relevant for the following three features. First, environments can be isolated from each other and are subject to hierarchical resource controls. Second, logic to control checkpointing and migration can be separated into an outer environment. Third, interposition gives an outer environment complete control over an inner environment's interactions with environments higher up the hierarchy, including one.world's kernel.

An environment appears to an application as a regular component. Each environment imports the event handlers called "*main*" and "*monitor*", and exports the event handler called "*request*". *main* must be linked to an applications' main component before the application can run in the environment and is used by one.world to notify the application of important events. Events sent to an environment's *request* handler are delivered to the first ancestral environment whose *monitor* handler is linked. The root environment's monitor is linked to one.world's kernel, which processes requests for structured I/O, REP and environment operations. Consequently, applications use the *request* handler for interacting with the kernel, and the *monitor* handler to interpose on all events sent to a descendant's *request* handler.

Implementation

one.world is not only an architecture existing on paper, it is also working. The researcher implemented one.world largely in Java because of its portability to a wide range of platforms. In addition to the parts in Java they use a small, native library to generate globally unique identifiers[3]—as they cannot be correctly generated in pure Java—,and the Berkley database to implement reliable tuple storage. The architecture currently runs on Windows and Linux PCs, and a port to Compaq's iPAQ handheld computer is under way.

2.6 EasyLiving

Of course, Microsoft Research cannot be out of exploring ubiquitous computing. In 1998 they started to work on a study called *EasyLiving* [EasyLiving00], which is now an operating system. They focus on an architecture and technologies for intelligent environments, that is why EasyLiving is the name of the project. Under intelligent environment they understand a space that contains myriad devices that work together to provide users access to information and services. Devices may be stationary or mobile. While the traditional notion of a PC is a part of their vision, a broader goal is to allow typical PC-focused activities to move off a fixed desktop and into the environment as a whole.

The researchers at Microsoft wanted to have an intelligent environment built from many different types of devices, like mices, keyboards, speakers, displays, active badge systems, cameras, wall switches, or even sensitive floor tiles. To support richer interactions with the user, the researcher found out that the system must have a deeper understanding of the physical space from both sensory (input) and control (output) perspective. For example, it might be desirable for the system to provide light for the user as he moves around night or make him coffee when user stops in front of a coffee machine in the morning. To sum up the foregoing, EasyLiving's goal was to develop an architecture that aggregates diverse devices into a coherent user experience. This required research effort on a variety of fronts, including middleware, geometric world modeling, perception and service descriptions. The attained issues will be shown in the following parts through the

[3]used by tuples

presentation of the architecture's components.

Generally two models of device integration can be used: peripherals in communication to a central machine or standalone devices in direct communication. Although both models provide complete solutions, the networked standalone devices are used at the basis for Easyliving because of the cost-effective computing power when they are in collaboration.

Given a collection of networked devices, the need arises for a mechanism that supports inter-machine communication. In spite of the fact that it already existed some middleware environments built on synchronous semantics, like DCOM, Java and CORBA, a new middleware solution was needed because of the failings of the older's. The two main failings were the forcing of programmers to employ a multithreaded programming model to avoid latencies inherent in synchronous communication, and the inefficiency of communication—even in a multi-threaded environment—because of the pipelining of messages between two endpoints.

Older middleware solutions were rejected because of a further reason: at each technology, processes describe the target of the message, consequently there are problems in message delivery when the target is moved to another machine. DCOM and Java require machine names as a part of the address for the message, CORBA provides for an object reference, but does not allow that reference to be updated dynamically. Since users frequently switch over laptops, desktops and home machines, and since mobile devices also often change both physical locations and network connectivity as they move with the user, it is indispensable for the client to have updated target address.

The middleware solution *InConcert* used in EasyLiving address the issues required for ubiquitous computing. InConcert provides asynchronous message passing, machine independent addressing and XML-based message protocols. By using asynchronous message passing, InConcert avoids blocking and inefficiency problems, and allow programs to handle offline and queued operation more naturally. Machine independent addressing is solved using *InstanceIDs*, that are unique names for components which are never reused. When started, a component requests an InstanceID and while running it provides a lookup service with a periodic keep-alive message. When InConcert is asked to deliver the message it will resolve the ID by asking the Instance Lookup Service for the current location of the instance. Once the message is delivered, the receiver decodes the content from the XML description. The receiver process only the information of the fields it can interpret. This is very useful, because messages from newer applications don't cause a failure to older applications which use older message syntax and can not interpret the additional fields.

Geometric knowledge, i.e. information about the physical relationships between people, devices, places and things, can be used to assemble set of User Interface (UI) devices needed for a particular interaction. The geometric world knowledge provides three capabilities:

- *physical parameters for UIs*: a User Interface can follow its user based on the information provided by the geometric model (e.g. a picture showed on a wall screen can be moved to a PDA's display when the user leaves the room where the wall screen is placed)
- *simplified device control*: the user must only focus on the task, the rest is done by the system (e.g. a user must not select a speaker when starting the music player, the system finds itself the nearest one)
- *shared metaphor*: the user does not need to know particular names of devices, the geometric model provides a shared metaphor between the system and the user, allowing a more natural interaction (e.g. when user wants to turn down a light, the provided map of the room allows him to quickly find and control the needed devices based on their physical location).

To locate a user or a device three methods can be used. *Outdoor beacons*, that are emitted by GPS and Cellular Phone-based locations systems are useful for outdoor location determination. Due to the physics of receiving beacon signals, *indoor beacons* must be used in buildings. In general, systems using indoor beacons consist of RF, IR or ultrasonic transceivers, which can determine the presence and perhaps location of small tags. These systems represent geometry as a location in a single coordinate frame, such as a map of building. *Network routing* can also be used to locate something: we can say that network or data connectivity is equivalent to co-location – if two devices can communicate directly they are co-located.

The *EasyLiving Geometric Model* (EZLGM) provides a general geometric service for ubiquitous computing, focusing on in-home or in-office tasks in which there are innumerable I/Os, perceptions, and computing devices supporting multiple users. The EZLGM is designed to work with multiple perception technologies and is aimed at geometry ”in the small”, that is, for applications which may require sub-meter localization of entities in the environment. The base item in the EZLGM is an entity, which represents the existence of an object in the physical world. The Geometric Model manage the position, the orientation and the physical expanse (as an extent in the geometric model using a polygon) of the entities, furthermore the relationships between them. Summarizing, EZLGM provides a mechanism for both determining the devices that can be used for a user interaction and aiding in the selection of appropriate devices.

In the majority the information provided to EZLGM is data gained from the world through sensors, that is, from physical sensing devices that are attached to computers running perception components. Such sensing devices can be very different. The EasyLiving project uses the following means and applications:

- *cameras*: they are used as a way of tracking the location and identity of a person or a movable device; at present the system consists of two color Triclops stereo cameras mounted high on the room’s wall and each connected to a PC.
- *Stereo Module programs*: they run on the PCs connected to the cameras and they process the registered color and depth images to produce information for the Person Tracker.
- *Person Tracker*: it is connected to the PCs running the Stereo Modules and its job is to integrate data from all the cameras. Furthermore Person Tracker manages a special area in the room called the ”person creation zone” (typically at the entrance of the room), where new instances of people are created and deleted.
- *RADAR*: it is an in-building location-aware system that allows radio-frequency wireless-LAN-enabled mobile devices to compute their location based on the signal strength of know infrastructure access points.
- *fingerprint reader*: with the use of a fingerprint reader a person can be identified

As mentioned, the system EasyLiving from Microsoft Research is working, however there are still a number of major issues to address – states the researcher. In the future the researcher plan to handle events (e.g. a CD player finished playing), build more robust lookup services that support discovery and scaling, extend the Geometric Model to support disjoint places, and add a new function to the system that enables the user to create and edit automatic behaviors.

2.7 Conclusion of the related work

To summarize the foregoings, there have been many systems designed in the field of ubiquitous computing till now. All of them have similar goals: to establish intelligent environments, wherein the devices seamlessly follow and support the tasks of the users. However in spite of the same goal, all the concepts approach the problem in different ways.

The project Aura acts as a proxy for the mobile user, that is it follows the resources of the environment with attention and supports the tasks of the user with the detected resources by capturing the users intention. Tasks are described in a platform-independent form, thus tasks can be initiated in different environments using different resources.

Project Oxygen emphasizes the use of speech and vision technologies. It reckons the devices as intelligent resources with them the users can communicate as they would interact with another persons. The concept defines fixed and mobiles devices, a software running on them, furthermore an intelligent network, called *N21*.

A third concept in the field of ubiquitous computing is the system one.world. The designer imagined an architecture that uses mobile code to support the migration of the users' tasks. When a user moves from an environment to another then its running task will be saved and moved to the new environment.

The researcher of Microsoft created a study called EasyLiving, which is a little bit similar to the project Aura. The aim of the concept EasyLiving is to aggregate diverse devices into a coherent user experience. For this purpose the members of the project did researches on middlewares, geometric world modeling, perception and service descriptions.

Chapter 3

The Blown-up middleware

3.1 The Blown-up concept

In the previous chapter of the thesis some ongoing complex projects were shown in the research area of pervasive computing. The *Blown-up concept* [BlownUp02]—that was first published and presented at the Student's Scientific Conference at Budapest University of Technology and Economics in 2002—addresses the goal to create a dynamically changeable, but uniform computing environment over an ad hoc network, integrating stationary and mobile devices that are wirelessly connected with each other. *Blown-up* gives a solution for connecting applications running on different devices together, thus supporting the communication of intelligent devices that are an indispensable part of ubiquitous computing.

Devices participating in the same computing environment form a so called *Personal Area Network*. Each application that runs on one of the devices sees the other applications and itself as they were all running on the same device. The Blown-up system sees the whole network as a single device, thus making the job of application developers easier. Using the Blown-up architecture, developers creating distributed applications over ad hoc networks do not need to take care of dynamically changing network topology. Furthermore they can use the uniform Application Programming Interface (API) provided by the Blown-up system.

Since mobile devices can move away from their partners anytime, Blown-up supports the dynamically joining and leaving of a PAN and therefore the dynamically binding and unbinding of applications. The binding procedure can be initiated and done manually as well as automatically. The Blown-up system gives an API for this purpose, the way of changing bindings depends on the environment: if the environment is intelligent and has some knowledge about the users' intention (e.g. through a camera, or other sensing device), then a special, so called control application can make automatic bindings, but if no such intelligent device is present, the user must do the bindings itself using the user interface of the control application.

The Blown-up concept claims that peripheral devices like mouse, keyboard, display, and soundcard are in wider sense applications too, because these devices can be imagined as special applications operating as a physical device. For example a keyboard is an application generating keyboard events. Thus, using this philosophy, devices and applications can be connected to each other, for instance a keyboard and a text editor. The concept do not distinguish a device from an application in aspect of bindings, both of them are called *services*. Moreover the denominations application, peripheral device and service are frequently used as synonyms.

Using Blown-up a nowaday used ordinary device can be divided, in other words blown-up[1] to special devices designed to run applications implemented for unique functions. For example a standard PC can be divided into a unique keyboard, mouse, display, cd-player, floppy-drive, main component and into a special mp3-player, text editor, web browser device. Special devices are consciously designed for a given purpose, thus they are more powerful and effective at the given field than an ordinary device.

3.2 Scenario

To better understand the Blown-up concept, an example scenario will be shown presenting the capabilities of the system.

In the morning Joe arrives in his office. Since he has worked on his Personal Digital Assistant (PDA) on the way to his office, he get tired of using his PDA's small keyboard

[1]the name of the concept originates from this perception

and small display. On his PDA he starts the control entity of the Blown-up system and looks up for available services in the neighborhood. He notices that his local wirelessly connectable monitor, keyboard and mouse is on, so he instruct the control entity to bind the local monitor, keyboard and mouse to his PDA enabling him to continue working on his presentation started on the train an hour ago. After the desired devices are bound together, Joe can continue his work easier using bigger input and output devices (Figure 3.1).

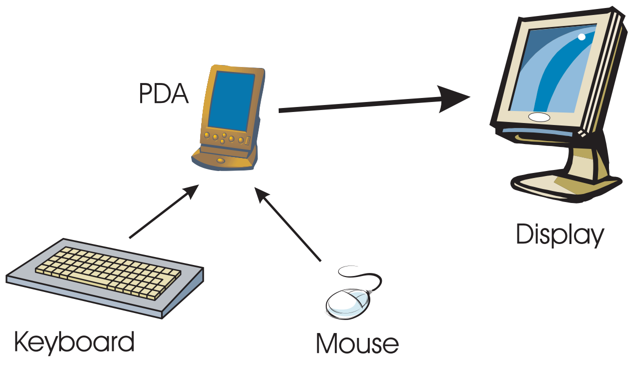


Figure 3.1: PDA bound with keyboard, mouse and display

Finishing the composition of the presentation, Joe wants to have some fun. He remembers that in the next room, by his colleague Steve, is a so called game machine on which a Tetris game is installed. Joe goes to Steve and ”borrows” the game, that is, Joe integrates the game machine into his own PAN. The game will be bound with the PDA’s small display of Joe and the wirelessly connectable keyboard in Steve’s room (Figure 3.2). After the successful binding procedure, Joe starts to play, but the small display disturbs him. He asks Steve to share the local display with him when Steve does not use it. Steve agrees, so when Steve is in the room Joe uses his own display integrated in his PDA, and when Steve leaves the room the graphics will be displayed on the bigger stationary monitor (Figure 3.2). The switching procedure can be done automatically as well as manually. In case of automatically switching, there are several sensing devices integrated into Joe’s PAN that perceive the presence of Steve, such as a camera or a voice recognizer. When there are no devices that are capable to sense Steve’s presence, Joe or Steve must manually switch between the used displays.

After a while Joe wants to listen to music, thus he redirects the control front-end of the

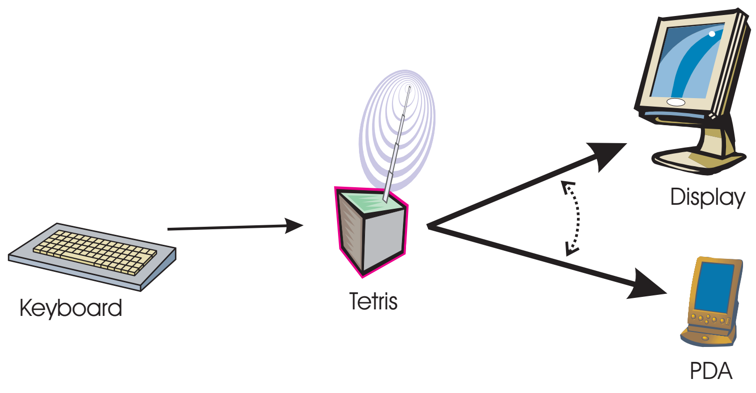


Figure 3.2: Tetris game bound with keyboard and display of a PDA

mp3-player placed on Steve's desk to his PDA. At the same time he binds the mp3-player with a file server placed in the corner of the room and with a speaker mounted on the room's wall (Figure 3.3). After all the bindings are created, Joe chooses a song using the mp3-player's front-end on his PDA and starts to listen to the music.

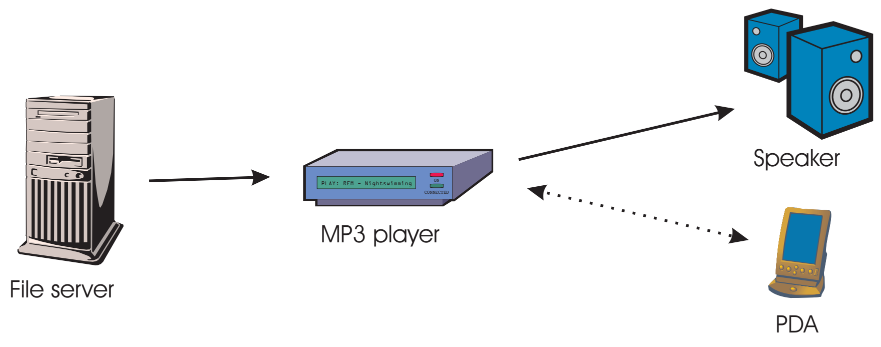


Figure 3.3: mp3-player bound with PDA, file server and speaker

Watching Joe, Steve wants to have some fun as well, therefore Steve asks Joe to play a two-player Tetris game jointly. They only need a second keyboard attached to the game device. Since Steve does not want to use his local keyboard that is currently bound to a text editor application, he decides to use the keyboard of his mobile phone. To create a common shared PAN to play two-player Tetris game, Joe first unbinds the existing two connections between the display, the keyboard of his PDA and the game machine, and then he creates three new connections: a PDA's keyboard to game device, a mobile phone's keyboard to game device, and a game device to local display (Figure 3.4). After all the bindings builded up, Joe and Steve start to play.

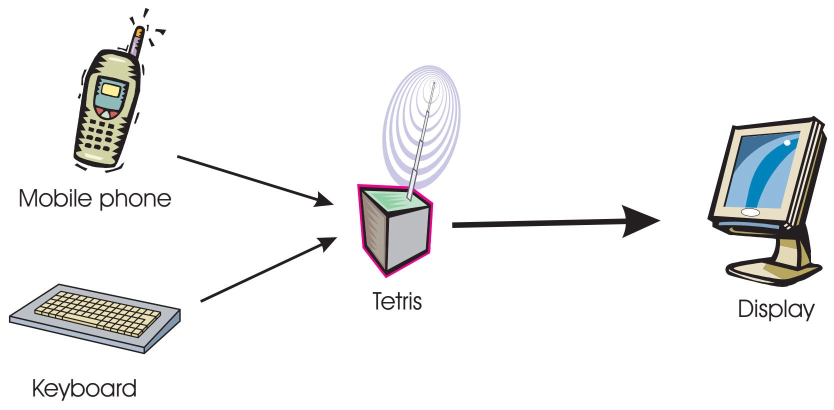


Figure 3.4: Tetris game bound with keyboard, mobile phone and display

3.3 Overview of the Blown-up system

3.3.1 Overview of the Blown-up architecture

The Blown-up system was designed as a *middleware* between the applications, the operating system and the hardware (Figure 3.5). On powerful devices the *Blown-up Micronet Protocol* (BUMP) may operate as a part of the operating system or above it, on limited devices—without operating system—(e.g. mouse, keyboard) directly reaching the hardware.

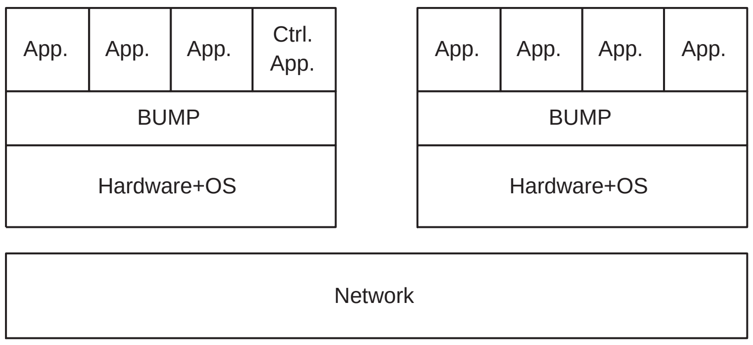


Figure 3.5: System architecture of Blown-up [BlownUp02]

In the Blown-up system the applications running on a device can connect to the services on other devices located in the Personal Area Network as they were all running on the same device (Figure 3.6). For example, in the scenario outlined in Section 3.2, the Tetris game can connect to the monitor located in the room and to the display of the PDA (i.e.

to both display services) as they were both organic part of the device the game is running on.

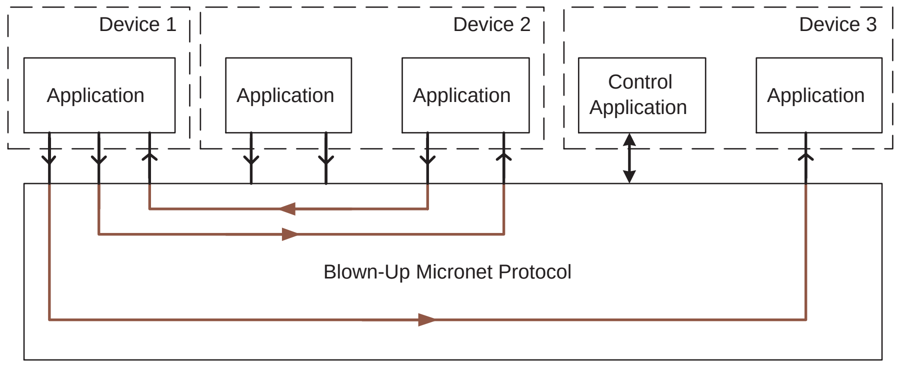


Figure 3.6: The world – as an application sees it [BlownUp02]

The services are presented in the system by their input and output access points, every service can be accessed through these outlets. If a service has a free output—e.g. the output of a keyboard or of a random number generator application—it offers to another service which can use it (which has an input of the same type); if a service has a free input—e.g. the input of a display or a video capture application—it makes it accessible. Blown-up looks at services as entities that requires structured data as input or output and their internal operation remain hidden. That is why applications are not distinguished from peripheral devices and are named jointly as services. These services can be bound together as the user or an application wishes.

Connections in Blown-up are point-to-point links, that is, two outlets take part in every connections, one of an input and another of an output. Every connection between an input and an output is ony-way in terms of data transport—since an output outlet is connected to an input (Figure 3.6)—, however in order to send and receive acknowledgments the link is practically bi-directional. In many cases more than one connection must be established at the same time in order to use a service (looking at the scenario in Section 3.2, the Tetris needs a keyboard and a display at the same time). The connections established—at the same time—to use a coherent group of services (e.g. the display, the Tetris game and the keyboard) are called on the whole *sessions*.

An output can be linked to more than one input as well as an input to more than one output at the same time. However data flows only at one of the links, on the link which has the so called *focus* on both sides of the connection. Focus can be switched using the *function changing focus*. This feature can the best explained with the "ALT-TAB" window switching in many operating systems: lots of applications run in the background, but only one window preoccupies the user's attention. Looking again at the example scenario in Section 3.2, a focus is changed when Steve leaves the room and Joe redirects the display output of the Tetris game to the monitor placed in Steve's room instead of continuing using the built-in display of his PDA.

Lots of applications and peripheral devices may offer many types of inputs and outputs to other services. Of course, a mouse-output should be bound to a mouse-input and there is no sense to bind an output of a random number generator to an input of a monitor. In order to bind only identical inputs and outputs, types were defined in Blown-up for inputs and outputs. Even an automatic type-control mechanism was designed to impede the linking of unidentical types of inputs and outputs.

The PAN needs at least one control entity that manages the connections. Although certain applications can locally have control rights—only in sessions they takes part—, only the main control entity has the rights to bind services together. For instance, this main control entity can be a PDA, which has enough computing power to manage a PAN. Toward the flexibility the Blown-up concept does not make restrictions in the number of control entities in a PAN, it allows the presence more than one's at the same time.

In the PAN of Blown-up manifold devices can operate together: desktop PCs, servers, notebooks, PDAs, mobile phones and specific devices made for a single purpose. This last can be for example a gadget, a game device, or a peripheral computer device (like mouse, keyboard, display) that possesses a network interface. Blown-up provides a solution for the developers to create distributed services for PANs over these heterogeneous devices as easy as possible, that is, only knowing and using the Application Programming Interface of the *Blown-Up Micronet Protocol*.

To bypass the misunderstandings about some notions, in the followings I describe the accurate meaning of some notion that are used in the specification of the Blown-up architecture.

By analogy to the notion used in the topic of ICs (Integrated Circuit), inputs and outputs of services are commonly named *pin*s. A pin is one-way (it is an input or output). A service may have obligatory and optional utilizable *pin*s, more about this will be written afterwards.

*Device*s are entities that offer services. The devices are distinguished using unique *Blown-up addresses.* The number of services running over a device is not limited, it depends on the device: typically a mouse will only offer a mouse service, but an ordinary PC can have more than one service to offer, like editing text, drawing picture or playing music.

The links between *pin*s are called *channel*s. Likewise *pin*s, in terms of data transport *channel*s are also one-ways.

3.3.2 Overview of the protocol stack

The Blown-up system uses the *Blown-Up Micronet Protocol*, which is composed of three layers: it contains a transport, a network and an adaptation layer. Above the BUMP are the applications, and under it the layers used for transmitting data. As it can be seen on Figure 3.7, the protocol stack is—additional to the three layers—vertically divided into two parts: into a **user** and a **control plane** (called as BUMP controller or BUMPC). The *user plane* is used by the applications for the purpose to effectively communicate with other services, the *control plane* is used in order to manage the connections and sessions.

Services connect through their pins to the modular built BUMP-transport layer (user plane). The pins send data to other services and receives data from other services through this layer. Each pin uses a given type of transport module depending on the requirements of the transport. When a control application wishes to create a new session or manage an existing, then it sends control commands through the control module located in the control plane.

The **BUMP-Transport Layer** is responsible for the transmission of data from a pin to another pin, guaranting the right order of the bits at the receiver and the fulfillment of the additional transmission requirements. Blown-up has different transport methods according to the type of the transmission: whether the data flow is a stream or a transaction operation in character, whether it is reliable and whether it uses flowcontrol. In addition, types are

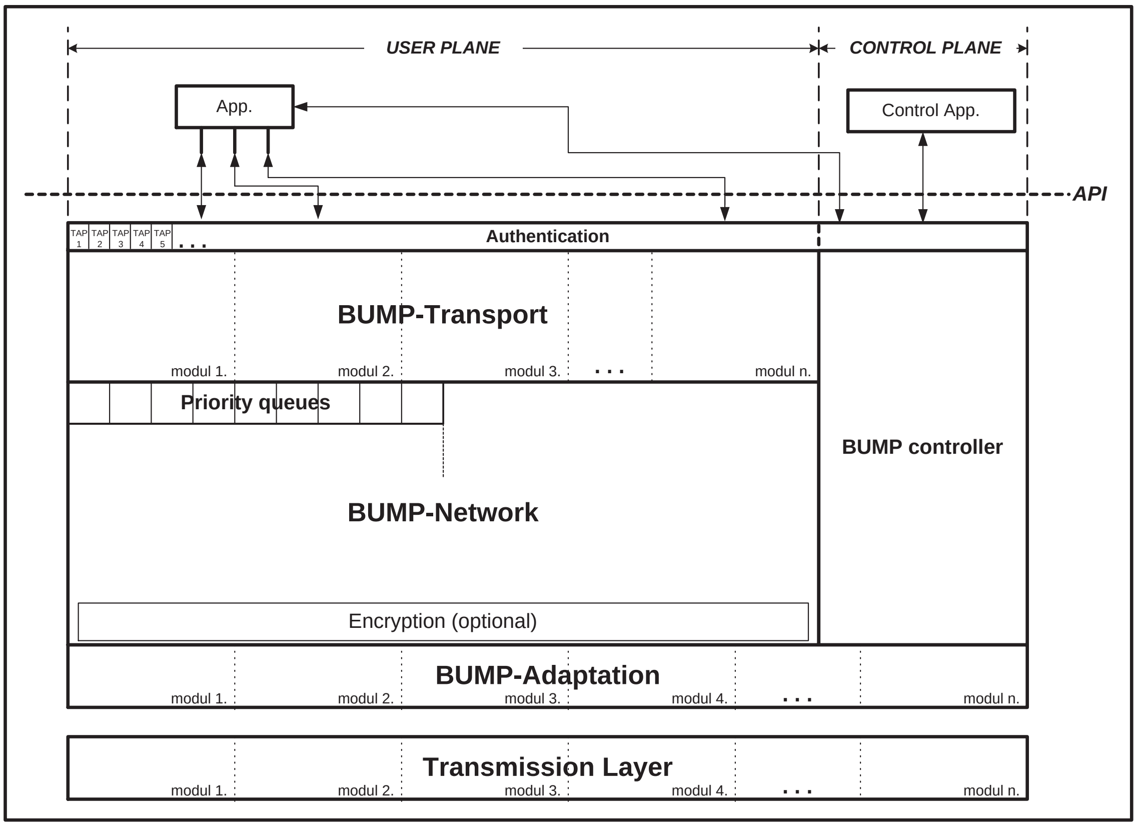


Figure 3.7: Protocol stack of the BUMP

defined based on the maximum size of data blocks the module can transmit.

The **BUMP-Network Layer** handles the channels: it registers the bindings between a local and another (local or remote) pin, and on the basis of this knowledge routes the packages of the BUMP-Transport Layer to the correspondent device. The BUMP-Network accepts messages from the BUMP-Transport Layer, transmits them to the corresponding BUMP-Network Layer that is on the other side of the channel and on this other side the BUMP-Network layer passes the messages to the BUMP-Transport Layer.

The channels in the network are created, deleted and managed by the **BUMP controller**. As described before, the channels are point-to-point links, however inserting a special service point-to-multipoint links can be simulated (with a service that has one input and few outputs of the same type as the input). In addition, BUMPC is responsible for service discovery.

The **BUMP-Adaptation Layer** adapts the BUMP to the transmission technology that the device uses to transmit data to the partner devices. The adaptation layer converts the BUMP messages to the right network format and in addition converts the internal BUMP addresses to the addresses used in the network.

3.4 The user plane

3.4.1 BUMP-Transport Layer

Each application that wants to communicate has one or more pins. These pins connect to the Transport Layer, i.e. to one of the Transport Layer's module through a Transport Access Point (*TAP*). The applications can choose for each of their pins the module to bind depending what type of channel they want to accommodate to the pin. At present three kinds of transport modules are designed with diverse attributes, but in the future some new modules can be added, for example a module that supports real time data transfer or a module that ensures QoS (Quality of Service). The attributes of the already designed stream[2], pipe[3] and block type modules can be seen in Table 3.1.

	stream	pipe	block
data unit	less than 100 byte	less than 100 byte	1 block
characteristic	flow	transaction	transaction
reliability	not reliable	reliable	reliable
flow control	speeding up / slowing down	contains	contains
data rate	low	low	high and bursty

Table 3.1: Stream, pipe and block type transport modules

The Transport Layer registers the pins of the applications. To each pin it stores his

- **owner**, that is the identifier of the service it belongs
- **priority** that indicates the privileges the data sent through this pin has during transmission
- **direction** that assigns the direction of the data flow traversing through the pin, thus giving if the pin is readable or writable (i.e. input or output)

[2]this can be compared to a brook, where water can ooze away
[3]this can be imagined as a closed pipeline, where all the water comes out that entered the pipe

- **state** that can be set to locked or not locked dependent upon the pin is ready to receive/transmit data or not

A pin is locked when a service uses it as an output and there are no channels bound to the pin, or when a channel is bound to the pin, but the pin at the other end of the channel is bound with several other pins and the focus is at one of this other channels.

When a service is connected to another service and sends data to this other application through one of his pins in form of an *appData* message (Figure 3.8) then the Transport Layer puts it into a *DATA* message. This *DATA* message includes the identifier of the TAP (*TAP-ID*) on which the data has been arrived and some additional special parameters required by the specific module, like frame number. The Transport Layer gives this *DATA* message to the Network Layer that sends it to the Network Layer of the partner device, which forwards the message to the corresponding local transport module. The transport module interprets the header of the message and based on this knowledge sends the body of the *DATA* message in form of an *appData* message to the adequate pin of the convenient service. In this wise data gets from an output pin of a service to the input pin of the partner service, which is on the other end of the channel.

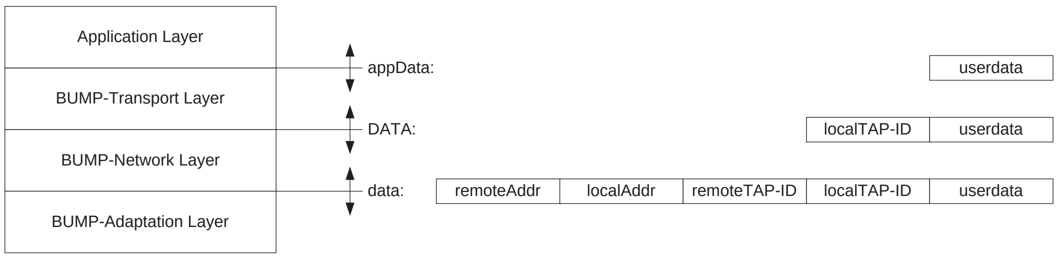


Figure 3.8: Forwarding data

At present it is only the pipe type transport designed in the details. The pipe type transport contains congestion control and uses selective retransmission to support a reliable data transfer over a noisy channel. If a data packet is lost then only this lost packet will be retransmitted. The packets following the lost one, however preceding it in point of arrival time, will not be thrown away, but stored until the lost packet arrives.

3.4.2 BUMP-Network Layer

The BUMP-Network registers the established channels (TAP-TAP joins) between two pins, contains several so called priority queues towards the adaptation layer and an encryption module to encrypt the messages sent through the network. The encryption module is not designed so far, only its location is given in the architecture.

The BUMP-Network Layer keeps a record of each channel connected to one of the local pins. Such a record contains the Transport Access Point (*localTAP-ID*) and the priority value (*Priority*) of the local pin, the Blown-up address (*remoteAddr*) of the device at the other endpoint of the channel, the TAP identifier of the remote pin (*remoteTAP-ID*), a variable that indicates whether the channel is locked or unlocked (is the pin that sends data through the channel locked or not) and the identifier of the session (*sessionID*).

As described above at the transport layer in Section 3.4.1, when an application sends data through one of its pins then the BUMP-Transport layer passes a *DATA* message to the network layer that comprise the data and the *TAP-ID* of the sender. The network layer interprets the header of this *DATA* message and on the basis of the *localTAP-ID* in the header and its own channel registry, the BUMP-Network layer decides to which channel the message should be forwarded. The network layer only forwards messages on channels that have focus. These are channels that connects to an input type pin thats focus is on the channel that wants to forward the message, since there is no sense to transmit the data if the receiver does not want to accept it. Once the network layer defined the channel that must be used to forward the received *DATA* message, it supplements the *DATA* message with the TAP identifier (*remoteTAP-ID*) and the device address (*remoteAddr*) of the remote pin that is connected to the opposite endpoint of the channel, thus generating the *data* message (Figure 3.8).

The just now created *data* message will be put into one of the output priority queues of the BUMP-Network Layer by the score of the priority of the pin. Each output priority queue has different priority level, so preference can be given to a message over other messages by the delivery. This is an indispensable requirement to the usage of the Blown-up system, since for example a user will not be happy if he could not use his mouse due the lack of bandwidth that the downloading of a bigger file has induced. The BUMP-Network gets the messages from the priority queues based on an algorithm designed for handling priority. At present many algorithms exist to support this function, but still yet there are not selected any of them.

At the receiver side, the BUMP-Network Layer has to pass the content of the *data* message to the BUMP-Transport Layer. To this it must only interpret the header of the message and send a *DATA* message to the transport layer that contains the identifier of the destination TAP (*remoteTAP-ID* in *data* message) and the data of the sender pin.

Summarizing the foregoing, the BUMP-Network Layer creates channels joining an input and an output TAP. Since the pins, and therefore also the TAPs are inputs or outputs, each channel is uni-directional in terms of data transmission. That is all very well, but as some transport module require acknowledgment and feedback about the procedure of the data transmission, channels are in terms of controlling two-ways.

3.4.3 BUMP-Adaptation Layer

The lowest layer of the BUMP, the BUMP-Adaptation Layer adapts the BUMP to the transmission layer that the devices use to transmit data over the network. Since the BUMP-Network Layer can be used in place of an ordinary network layer, in the Blown-up concept datalink layer and physical layer are common referred to as transmission layer (e.g. TCP/IP, WLAN, Ethernet or Bluetooth).

The BUMP-Adaptation Layer is composed of one or more modules, where each module realizes an adaptation to a special transmission layer. Nowadays IP is so widely prevalent that it is obvious to use it in the first place for transmission layer, but contains many functions that the BUMP does not need and this way its usage could needlessly complicate a slim device that provides simpler services (e.g. a mouse service). Furthermore some additional adaptation modules can be implemented that supports other communication technologies, like WaveLan (IEEE 802.11) and Ethernet direct access, or Bluetooth using the piconet feature of this latter technology to dynamically build groups of devices.

In consequence of the foregoings, the BUMP-Adaptation Layer has two main role: it sends and receives BUMP messages as specified by the transmission layer and converts internal

Blown-up addresses into addresses used by the transmission layer.

3.5 The control plane

3.5.1 Overview of the BUMP controller

The BUMP controller provides the tasks of the control plane. These tasks are:

- advertising of the services offered by the applications on the local device,
- looking for and collecting the services offered by other devices in the PAN and—on demand—sending this knowledge to a local control application (service discovery),
- binding of local pins to local or remote pins in order of a remote bump controller,
- creating sessions, managing them and—after use —unbinding the connections on request of a local control application or a local application that has control ability.

Any of the applications running on a device can send commands to the local BUMPC. These commands can instruct the BUMPC to register or unregister an application and to create, manage or tear down a session. The control applications are able to manage their own sessions (sessions created by the given control application), while user applications can only modify a session if at least one of their pins participates in the session, but in such cases they have only limited rights. Thus, while a special control application is able to bind pins on remote devices, a simple user application can't do this. The rights of the applications are managed and controlled by an authentication module above the BUMP controller: it registers all the applications and their rights, furthermore checks the requests of the applications based on this knowledge.

The BUMPC handles several tables to register the local and remote applications and to manage the connections in the PAN. These tables and their roles will be described in the following sections when presenting how an application registers and unregisters itself, and how connections are built, managed and torn down.

3.5.2 Registering a user application

The applications using the BUMP have to have at least one input or output pin. To send or receive data through these outlets, the applications have to be registered by the system. Using the control messages sent to the control plane—that will be presented in this sections—the user applications can inform the system about the pins they have.

The BUMPC has to have the following knowledge about a service:

- the **name of the service** (*AppName*) that, complemented with the address of the device, unequivocally identifies the service for other services in the PAN;
- the **description** of the service (*AppDescr*) that contains additional information about the service (for example the function of the service) compared to the name of the service;
- the **internal identifier** (*AppID*) of the service;
- **properties of each pin**, which are the followings:
    - the **type of the pin** (*PinType*), that denotes what kind of data it transmits (for example the type of a mouse's pin can be *type mouse events*). The declaration of the pin types makes to the system possible to easily check whether a control application wishes to bind pins of the same type (for example it would have no sense to connect the output of a mouse to the input of a display);
    - the **direction of the data stream** (*I*nput / *O*utput) that indicates whether the applications wish to read or write the pin. Simultaneously a pin can have only one direction, if the service wants duplex transmission it has to use two pins;
    - the **necessity** of the pin (*O*ptional / *M*andatory) to signify whether a service can only be part of a session when the given pin is used, that is, a channel is connected to it, or also without connecting it to another pin. Looking at the example scenario in Section 3.2, it would have no sense to play with a Tetris game that does not have any display connected to whereat it could show the graphics of the game;

- in case of an output pin, the **priority** (*Priority*) of it. This becomes a role when service sends data: as described by the Network Layer in Section 3.4.2, data sent through certain pins has priority in contrast to other;
- the **capacity** of the pin (*Capacity*) that indicates the maximal number of the channels that can be bound to the pin at the same time. Data can always sent through the channel that actually has the focus;
- the **identifier of the transport layer module** (*TpType*) used by the pin to send or receive data;
- the **internal identifier** of the pin (*TAP-ID*) that identifies the Transport Access Point to which the pin is bound;
- the **outer identifier** (*MyPinName*) of the pin that was given by the user, unambiguously identifying the pin for other users. In contrast to the internal identifier, outer identifier is a text string, since such a representation of a pin is more expressive for a user as a number;

BUMPC stores notes containing the properties described above in so called service-registry tables.

The process of application registration can be best compared to transactions used by databases. As the first step the application sends a *Hello* message to the BUMPC that contains its identifier, name and description (Figure 3.9). After this, the application requests the registration of its pins: it sends so called pin registration messages (*RegPinReq*) to the BUMP controller, each of them containing one of the pin's properties. If the BUMPC successfully registers a pin then it sends *RegPinResp* message back that includes the *TAP-ID* of the newly registered pin. After that the application registered all its pins, it directs a *RegFin* message to the BUMPC, which signifies the end of the registration process. The BUMPC adds the new service to its service-registry table and from that time on it periodically advertises this service with *beacon* messages for other BUMP controllers in the PAN. Such a *beacon* messages comprise the Blown-up address of the device, the name and the description of the service.

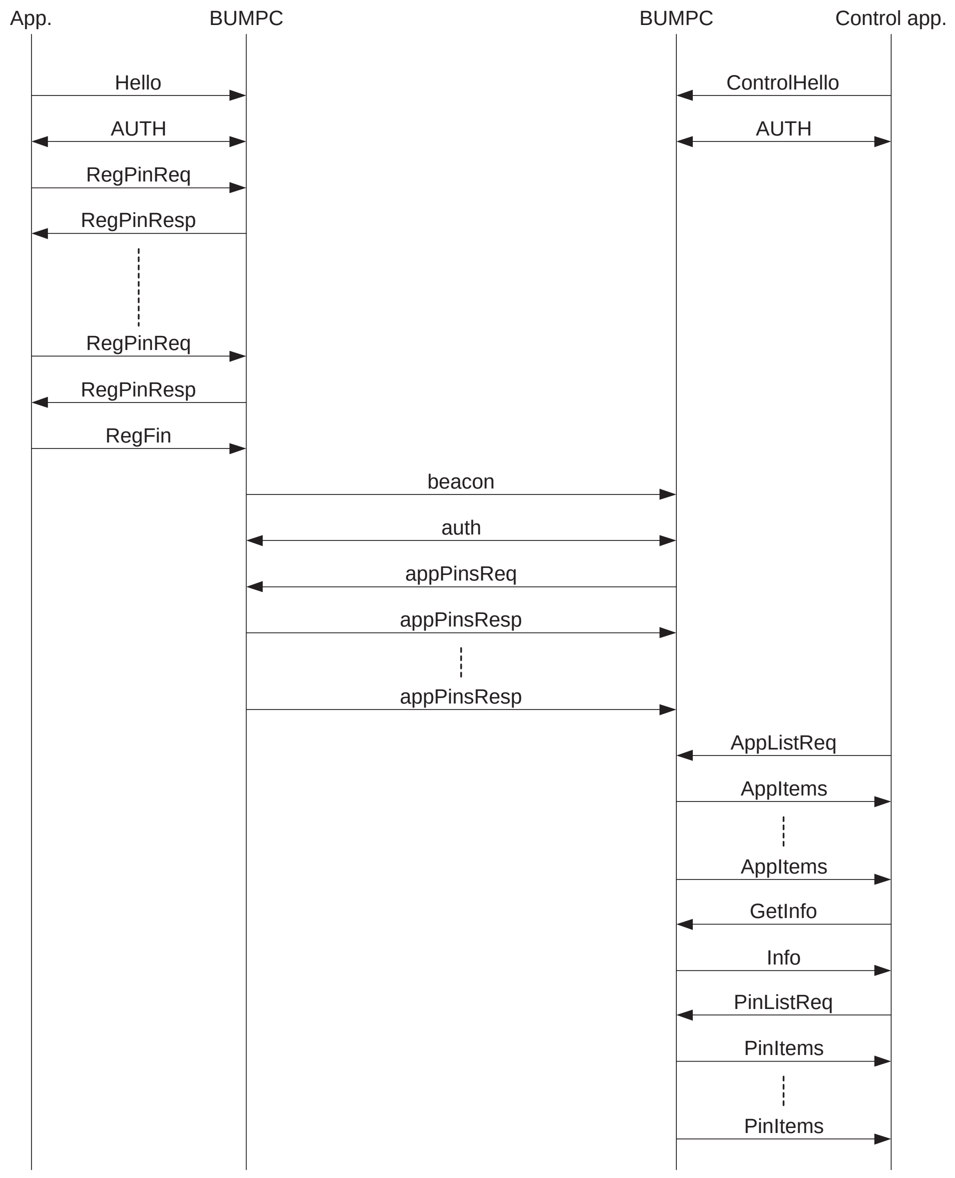


Figure 3.9: The registration, advertising and querying of an application

BUMP controllers in the PAN continuously scan for *beacon* messages. When a BUMPC receives a new advertising message then it sends an *appPinsReq* back to the sender of the *beacon* message asking for the pins of the newly detected service. In answer to this request, the BUMPC on the device of the new service returns the properties of the pins in one or more *appPinsResp* messages, each pin's properties in a separate message. Thus a BUMPC become aware of a new service in two steps (new *beacon* and then some *appPinsResp* messages) saving significant bandwidth, since the properties of the pins are sent only once per device and so the frequently emitted *beacon*s contain only basic information about the services. The information about the services on other devices is stored in the service-registry and passed to a control application on its demand.

The BUMPC continuously checks whether it periodically receives the *beacon* messages from the known services. If not (a timer expires that was reseted when receiving the last *beacon*) then the BUMP controller removes the application from its service-registry. The stopping of the beacon's reception can occur in two cases: if the given service is revoked (the application has been unregistered) and the BUMPC ceased to send beacons about the service, or if one of the mobile devices moved away, thus they are not any more in radio range.

3.5.3 Creating a new session

A session can be established by a control application or by a user application that has control abilities. Looking at the example scenario outlined in Section 3.2, the PDA has such a role: it creates a session binding the Tetris game to a display and to a keyboard.

When starting a control application it has first to register itself at the BUMP controller by sending a *ControlHello* message to the BUMPC and in addition some other authentication messages required by the actual used authentication method (Figure 3.9). After the successful registration, the control application can query the services offered in the PAN passing an *AppListReq* message to the BUMPC. In answer to this query message, the BUMPC receives *AppListResp* messages that contain only the basic information of the available services: each message includes the information found in the body of a beacon. If a controller applications wants to know more about a given service it must send a *GetInfo* message to the BUMPC whereupon it receives an *Info* message that contains the description of the queried service. The pins of an application and their properties can be retrieved by sending a *PinListReq* message to the BUMP controller indicating the interested service. In answer the BUMPC will send *PinItems* messages: one per pin.

Establishing of a session happens—like the registration of a service's pins— as a transaction. Before sending any messages, the control application selects a unique (within the application) transaction identifier (*transactionID*) that can be used by itself to unambiguously denote its own session. Then the control applications passes a *TStart* message to the

BUMPC announcing the beginning of a new session establishing procedure. Of course, the message contains the *transactionID*. Based on this transaction identifier and the internal identifier of the application (*AppID*) the BUMPC creates a *sessionID* that completed with the Blown-up address of the device, will be unique in the PAN. This latter identifier will be used in the network to identify a session created by a controller application.

After sending *TStart* the controller application sends one by one the whished pin-pin bindings to the BUMPC in form of *ConnPins* messages. It closes the list of requests with a *TEnd* message. The formation of the session failed if it receives a *TNOK* message in answer from the BUMP controller, however a *TOK* message in answer means the session was successfully created. In case of a positive acknowledgment, the *TOK* message contains which optional pin–to–optional pin connections were really created, since a session can be successfully established without such bindings too. When creating a session the BUMPC requires only the successful establishment of channels that have at least one endpoint that must be connected to a mandatory pin. If a connection of two optional pins failed in the course of creating a session then the user can choose whether she or he wants to use the session without the desired channel or to tear the session down and perhaps try to create a new one.

As mentioned above, BUMPC begins to create a new session when it receives a *TStart* message from a control application. As the first step it checks each desired connection if it is permitted, that is whether the user wishes to bind an input pin with an output, and whether the pins are of the same *PinType* and *TpType*. If all wanted connections are allowed and the local PINs which will be connected have free capacities then the BUMPC sends for each remote bindings (i.e. for each channel that will have at least one endpoint on a remote device) a *bind* message that contains in its body the address of the local device (*controlAddr*), the *TAP-ID* and device address of both pins and the *sessionID*. If the connection of a pin-pair fails then the BUMPC returns an error by sending a *TNOK* message to the controller application.

The BUMP differentiates two types of *bind* messages in terms of interpretation and processing. By the first type, the sender of the *bind* message corresponds with the device of one of the required channel's endpoint, by the second type the *bind* message does not contain any TAPs that are on the sender device, i.e. in this latter case the BUMPC of a device binds TAPs of two remote devices. The recipient of the *bind* message is always one of the channel's endpoints:

- if the channel must be established between a TAP on the device of the sender and a TAP located on a remote device, then implicitly the remote device
- if the desired channel will not have any endpoint on the device of the sender, then the recipient of the *bind* message is one of the channel's endpoints, no matter which of them.

If the sender of the *bind* message is one of the channel's endpoints and the receiver of this message has the desired TAP and has moreover free capacity, then the recipient answers with a *bindACK* message to the query and at the same time it refreshes its table of connection requests (the next paragraph will explain the role of this table). Otherwise—if one of the conditions does not fulfill—the recipient answers to the sender with a *bindNACK* message. The recipient of the answer also refreshes its own table of connection requests based on the positive or negative acknowledgment. However, if the sender (the device of the sender) of the *bind* message does not take part in the connection, then if the recipient of the message has the denoted pin and is moreover free then the latter one has to forward the request to the other endpoint of the channel. Thus the recipient of the forwarded *bind* message becomes aware of the required connection and will act as already described, since it becomes a *bind* message from an endpoint of a channel. If the latter sender receives a *bindACK* from the third party then it refreshes its own table of connection requests and forwards the positive acknowledgment to the device of the controller, otherwise sends a *bindNACK* message back (Figure 3.10).

The BUMPC stores the received connection requests in the above mentioned connection requests-table assigning a state to each request. This state can be "pending" or "runnable". When the BUMP controller receives a new, so far unknown *bind* message and the pin appointed in it is free (i.e. has free capacity) then it appends the request to the connectionrequests-table with the state of "pending". This record will have this state until all mandatory pins that belongs to the same application receives a request from the same control device with the same *sessionID*. When this occurs then all records that holds requests to this preceding local application with the preceding *sessionID* becomes the state "runnable".

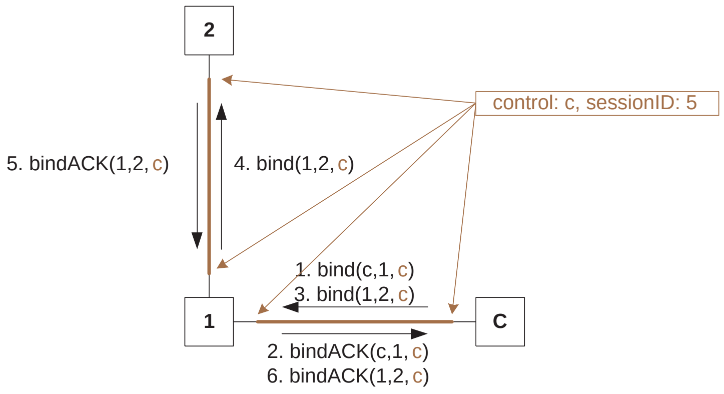


Figure 3.10: The BUMP controller on device c creates a new session [BlownUp02]

From that time, on the new requests that originates from the preceding device, moreover contains the preceding *sessionID* and wants to connect an optional pin (as all mandatory pins are already connected) of a preceding application, will be registered with the state ”runnable” in the connection requests-table.

After sending the *bind* messages, the BUMP controller on the device of the control application waits for the acknowledgments, one per *bind*. If till a given time it do not receives all the acknowledgments for the request of channels that should be connected to at least one mandatory pin, then it sends the *bind* messages several times again. However it stops taking efforts after certain number of tries without effects. In cases of such a failure BUMPC passes a *TNOK* message to the control application.

Immediately after receiving the last acknowledgment for the bind requests that contained at least one mandatory pin, the BUMP controller broadcasts a *run* message to the devices in the PAN indicating that the session must be fixed, i.e. that the session must operate; from this time on applications can send data through the newly established channels. The other BUMPCs in the network receives this messages and inspects their connection requests-tables: if they have a record with the indicated *sessionID* and the state stored in the record is ”runnable” then they pass the record to their own BUMP-Network, which refreshes its registry of channels based on this new entry and in case of a pin’s first connection it gives the focus to the pin. Furthermore the devices send a *runACK* back to the sender, one per each local application that participates in the new session.

The BUMPC, that sent the *run* message, waits for as many acknowledgments as many application in the session take part. When all the have *runACK*s arrived then the BUMPC sends a *TOK* to the control application indicating the successfully establishment of the session. If till a given time it do not receives all the acknowledgments then the BUMPC resends several times the *run* message, but after certain number of attempts it do not retries any more to receive the *runACK*s, however sends a negative acknowledgment to the control application about the unsuccessfully session establishing process.

As described above, the Blown-up concept establishes a session in two steps, thus avoiding the unnecessary allocation of a resource. In essence the first step (*bind*s and their acknowledgments) is used to check whether all channels can be built in the session and only the second (*run* and its acknowledgments) step creates them.

3.5.4 Tearing down a session

The tearing down of a session can be initiated by:

- an application that's pin takes part of a session;
- a transport layer that transfers the data of the pins;
- a BUMPC that has at least one local TAP that participates in the session
- the BUMPC that has created the session.

However only the BUMPC that has created the session has the right to tear down the session, so when one of the first three entities mentioned above wants to end the session then it must send a request to the session creator BUMP controller.

If an application wants to revoke the service it offers, it has to send a *RevokeApp* message to the local BUMPC (Figure 3.11) which must contain its internal identifier (*AppID*). The BUMPC, after receiving this request, sends *unbindme* messages to the BUMPCs that have previously connected one or more channels to local TAPs belonging to the revoked application. The *unbindme* contains the identifier of the session (*sessionID*).

The BUMPC that receives the *unbindme* message answers to this message with a broadcasted *unbind* message, thus starting to unbind the session, since there is no sense to use

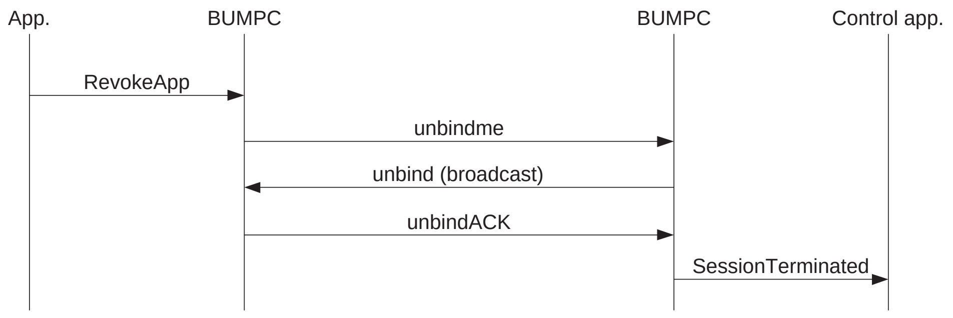


Figure 3.11: Revoking an application

a connection any more without a required service.

All the devices in the PAN receive the *bind* message and check their tables whether they have requests or already built channels with the received *sessionID*. If yes, then the BUMP-Network Layer and the BUMP controller removes the entries from their table that contains the preceding *sessionID*, thus tearing down the channels that took part of the indicated session. Furthermore the preceding BUMPCs send an *unbindACK* message back to the sender of the *unbind*.

The BUMPC, that sent the *unbind* message, waits for as many acknowledgments as many application in the session took part. If till a given time it does not receive all the acknowledgments then the BUMPC resends several times the *unbind* message, but after certain number of attempts it does not retry any more to receive the *unbindACK*s. In both cases, after all BUMPCs have answered to the *unbind* request, and after the timer expired several times, the BUMPC regards the session as torn down, therefore removes all its entries associated to the certain session and sends a *SessionTerminated* message to the control application. The remote BUMPCs in the PAN that have not received the *unbind* request and have one or more channels associated to the preceding session, will remove their entries automatically by detecting the absence of their partners.

A session can also be torn down by the creator of the session. In this case, the control application has to send a *SessionEnd* message to the local BUMP controller (Figure 3.12) indicating its internal identifier (*AppID*) and the *transactionID* in the message, which identifier was used to create the session. From now on everything happens like previously presented, as if the local BUMPC had received an *unbindme* message.

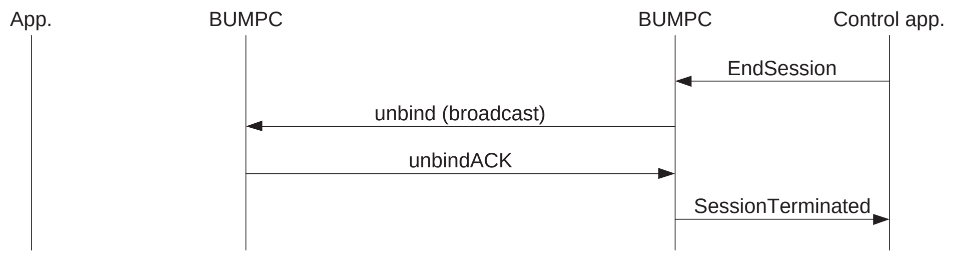


Figure 3.12: Tearing down a session by the creator

A transaction layer tears down a connection when the channel is unusable since a longer time, for example when the layer sends data through a channel without avail, i.e. none of the acknowledgments arrive. In such cases the transaction layer passes a request to the local BUMPC to end the session, which BUMPC then sends an *unbindme* to the creator of the given channel.

The tearing down of a session—as mentioned above— can also be initiated by a BUMPC that has at least one local TAP that takes part of the session. BUMPC continuously checks whether all the remote endpoints of the channels attached to the local TAPs are available or not and when it detects the lack of a connected device's beacon, the BUMP controller requests the end of the session by sending an *unbindme* message to the session creator BUMPC of the session.

BUMPCs that have sent an *unbindme* message, but have not received an *unbind* message in answer to the request within a given time, try to resend the *unbindme* message. They retry several times, but stop the attempts after certain unsuccessful essays and locally remove all the entries about the given session.

3.5.5 Changing focus

At the same time more than one channel can be bound to a pin of an application, but data can only be transmitted through one of them, which has the focus. Switching between the channels bound to the same pin—to select the actually used one—can be done generating a focus change event. Such an event selects the next channel that is connected to the given pin from the list of channels and makes it active in terms of data transfer.

The focus can be changed by local BUMP controllers or by remote BUMP controllers that created a session the pin takes part. The former one has adequate rights to change focus only on local pins, however the latter one can change the focus on remote and local pins that are bound to channels that were created by itself.

the local BUMPC. This message must contain the identifier of the application and the identifier of the pin (*AppID* and *MyPinName*). Receiving such a message the BUMPC first checks if it is possible to change the focus (more than one channel is bound to the pin), if yes, then it gives the focus to the next channel connected to the pin. If the pin bound to the remote endpoint of the previously used channel is an output pin then in addition to the locally switching procedure the BUMPC sends it a *lockOutputPin* message that warns the receiver not to send any data more on the channel, because the input pin connected to the remote endpoint of the channel is busy. When the receiver accepts this message then it locks its given pin, i.e. it does not send data until it receives an *unlockOutputPin* message. If the BUMPC that changed its focus has sent an *lockOutputPin* message to the endpoint of the previously used channel then it must send an *unlockOutputPin* message too, now to the endpoint of the actually used channel, indicating the new state of the pin: it is not any more locked (looking at Figure 3.13 the device number 2 changes focus locally from the channel between devices 1 and 2 to the channel between the devices 2 and 4, and further to the channel between devices 2 and 3)

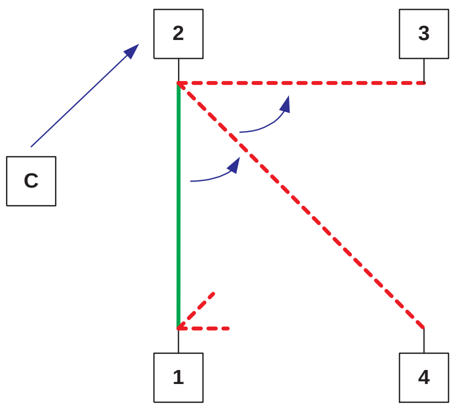


Figure 3.13: Changing focus [BlownUp02]

As mentioned above, focus can be changed on remote devices too. In this case a remote BUMPC has only the rights to change focus on channels that were created by itself and only when the focus is on an own channel. This means that a focus can only be given over to a next channel, focus can not be acquired from a channel of another session. To change focus such a way the control application has to send a *ChangeRemoteFocus* to the BUMP controller, which forwards the request to the remote BUMPC in form of a *chfocus* message. On the remote device the focus is changed as already described, however after the successfully procedure an acknowledgment is sent back to the requester. For example, looking at Figure 3.13, the control device changes the focus of a pin on device number 2: it switches the focus from the channel between devices 1 and 2 to the channel between the devices 2 and 4, where the channel between devices 1 and 2 belongs to a session that was created by the BUMPC of the control device.

3.6 The application programming interface

The Blown-up system has an Application Programming Interface to enable programmers the easy usage of the BUMP. Functions are defined which are collected in a library. The programmer must only include this library to its application and can directly access the system. The functions are classified into two parts: there are functions for user programs and other functions for control applications. In the following I will present the functions provided by the API, but won't get into details – these can be found in the appendix.

3.6.1 User functions

User functions are API calls, which can be used to register and unregister an application, furthermore to send data to and receive data from a pin. These user functions are the following:

- `bump_RegisterServiceStart`: it is used to begin a registration of an application. The user must pass the name and the description of the application as parameter.
- `bump_RegisterPin`: it must be called after the `bump_RegisterServiceStart` call and it serves for registering the pins of the application. Each function enrolls one pin. It requires among other the outer identifier (that is used in the future by the user) and the type of the pin, and gives back whether the pin was successfully registered.

- `bump_RegisterServiceEnd`: it must be called after the last pin-registration function and indicates the end of the application's registration. It gives back whether the application was successfully registered or not.
- `bump_RevokeService`: it is used to revoke a service, typically when exiting from an application.
- `bump_SendData`: with this function it is possible to send data through a pin, but only if a channel is connected to the given pin. In addition to the data, the user must give by each call the length of the data and the name of the selected pin. It returns the following values:
  - -1, if the connection does not exist
  - 0, if the pin is locked (the remote focus is on another channel)
  - 1, if the data was sent
- `bump_ReadData`: it serves for receiving data from another pin. When invoking it, the user must give the name of the certain pin to use, furthermore the place and the size of the memory area reserved for the new data. It gives back the length of the received data.

3.6.2 Control functions

Control functions are API calls that serve for creating, managing, or tearing down a session. The following functions belongs to this group:

- `bump_RegisterControlEntity`: like user applications, control applications have to register themselves by the BUMP controller as well. This function does the registration of a control entity and gives back whether the process was successful.
- `bump_GetServicesReset`: this API call is the first step to query the services offered in the PAN. When calling it, the BUMPC goes to the first entry in the list of services that were detected in the PAN (see also `bump_GetServices` to better understand the role of this API call).

- `bump_GetServices`: it is used to get the services located in the PAN. Each call gives information (identifier and name) about an additional service until the user queried all the services. Before the first call, the control application must call the `bump_GetServicesReset` to signal the BUMPC that it should give the first registered application back by the next query for application.
- `bump_GetInfo`: this function gives back the description of the application given as parameter to the function.
- `bump_GetPinsReset`: this API-call is similar to the `bump_GetServicesReset` function; it goes to the first element in the list of pins that belong to the application given as parameter.
- `bump_GetPins`: using this function the control application receives the pins and their properties of the application it gave as a parameter to the function. Like `bump_GetServices` it returns the pins of an application one by one and the control application has to call the `bump_GetPinsReset` function before the first query for the pins of an application.
- `bump_CreateSessionStart`: as described earlier, the creation of a session is conducted as a transactional process – this function initiates the transaction.
- `bump_BindPins`: it is used during the creation of a session to give a desired channel. The control application must indicate a pair of pins (i.e. two device addresses, application identifiers and pin names) that should be bound together. The control application has to call this function for each desired connection.
- `bump_CreateSessionEnd`: this function call ends the transaction started to create a session. If the session was successfully created then it gives a positive acknowledgment back, otherwise a negative.
- `bump_ChangeLocalFocus`: it is used to change focus on a local pin, the pin must be passed as a parameter.
- `bump_ChangeRemoteFocus`: this function—in contrast to the previous call—changes focus on a remote device.

- `bump_UnRegisterControlEntity`: it serves for unregistering a control application, typically called when a control application exits.

Chapter 4

Implementation of the Blown-up middleware

4.1 Introduction to the implementation

In the previous chapter I have presented the Blown-up concept and the Blown Micronet Protocol that is used to connect services in a Personal Area Network. As part of my thesis I implemented the main parts of the BU system. In this chapter I will present the implementation of this system and the process how I have tested it.

I have developed the Blown-up system for Linux operating systems in C++ language. The implementation runs in the user space as a user process, so it is not part of the kernel. The BU entities which are placed on different machines are communicating with UDP/IP protocol. The communication between the different BU systems and applications (both control and user applications) is realised via Inter Process Communication (IPC) calls. For device-device connections each system uses one UDP port on each device, for system-user program connections there is one IPC queue for each pin of an application, one single IPC queue for the BUMPC, furthermore one for each program. These latter channels are used to send control messages to BUMPC and to the programs (for example acknowledgment messages).

Like Unix and Linux handle devices and processes as special files, the pins of the applications are represented in the filesystem too. There is a root directory named `bump`, which contains subdirectories for each application. Each name of a sub-directory is an identifier of an application (*AppID*). The pins of the applications are represented by files located in this directories: each file of a pin is located in the directory of the application it belongs, and is named like the pin is called by the user (outer identifier of the pin, i.e. *MyPinName*). The identifiers of the queues used by IPC are generated based on this files, since C++ contains a function (`ftok()`) that can generate a unique and always the same identifier by a name of a file and its location in the filesystem. For the identifiers of the control queues the name of the application's directories are used, and the root-directory `bump` for the queue identifier of the BUMP controller. For example if the BUMPC has to send data to the pin called "displayinput1" of the application that has the iternal identifier 834, then it determines the identifier of the adequate IPC message queue based on the file "bump/834/displayinput1".

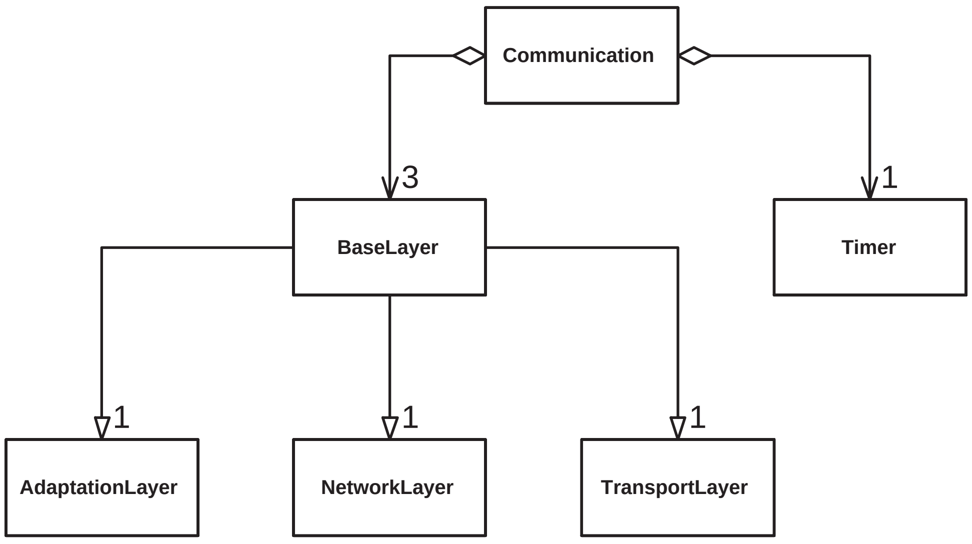


Figure 4.1: The diagram of the classes

My system is built of five significant classes (Figure 4.1). The three layers (BUMP-Network Layer and BUMPC together, BUMP-Adaptation Layer, and BUMP-Transport layer) are placed in separate classes and are all derived from a base class (*BaseLayer*) that realises a universal layer. In addition there is a class to connect this three layers and to support their communication (class `Communication`), and there is another class to handle timings (class `Timer`). The functions of the API—that help programmers to develop their BU-applications—are placed in a separate library.

In the following I will present the main attributes and functions of the classes and the API-library (their detailed operation is described in Section 3.4 and Section 3.5).

4.2 Class Communication and class Timer

The class `Communcication` (*CommC*, Figure 4.2) is the entity that connects the three BUMP layers, detects new messages sent by a BUMP system of another device or by an application, and handles timings. If new message arrives from the outside world, then it calls the receive function of the appropriate layer, and calls the call-back function of a timer if the given timer expires.

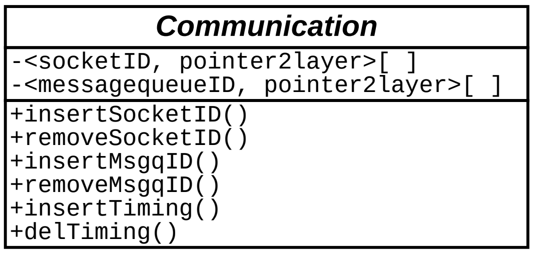


Figure 4.2: The class Communication

The class has to continuously check whether a new message has been arrived at an UDP port or an IPC message queue, therefore the class needs to know the UDP ports and the message queues used by the layers. For this purpose the CommC contains two lists: one of <UDP socket identifier, layer pointer> pairs and another of <IPC message queue identifier, layer pointer> pairs. In each pair the layer pointer indicates the layer that must receive the message sent to the certain UDP port or to the certain message queue. When a layer opens a new UDP port or creates a new IPC message queue then it has to call the `insertSocketID` or `insertMsgqID` function of the class `Communication`, which inserts both the identifier of the new socket or queue and the identifier of the creator layer (i.e. the identifier of the layer that called the function) to one of its lists. The messages sent in the future to the UDP ports or to the message queues will be passed to the right layer based on these two lists. Deleting an entry from this list can be done calling the `removeSocketID` or `removeMsgqID` function.

When the CommC detects a new message at an UDP port or at a message queue then it looks into the lists, and selects the layer that must receive the newly detected message. After found out which layer is the recipient of the message, it calls the `receiveMessage` function[1] of the specified layer.

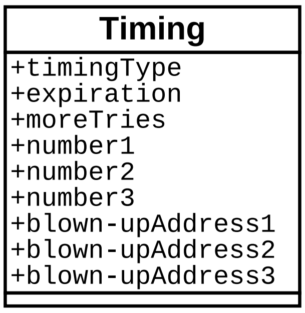


Figure 4.3: The class Timing

In addition to detecting new messages, the class `Communication` has to handle the timers used by the Blown-up system. I have created two classes for this purpose: the class `Timing` (Figure 4.3) realises a single timer, the class `Timer` collects all the `Timing` entities. The class `Timing` contains:

- a variable that indicates the type (*timingType*) of the timer, which is used to distinguish the different timers;
- the expiration of the timer (*expiration*);
- a counter (*moreTries*) that denotes how many times can the timer expire, that is how many times can be the call-back function recalled before the timer irrevocably expires;
- the variables *number1*, *number2*, and *number3* used for special aims, their meaning depends on the type of the timer (for example if the timer is associated to a call-back function that needs the identifier of the session as parameter then one of the variables denotes the *sessionID*);
- three Blown-up addresses that are used for special aims too, and like the three previous mentioned variables their meaning depends on the type of the timer.

[1]the role and operation of this function will be presented in Section 4.3, in Section 4.5 and in Section 4.4

The class `Timer` includes a sorted list of `Timing`s and several functions to insert and remove timings from the list.

If a layer wants to set a timer it calls the function `insertTiming` of the class `Communication` with a class `Timing` as parameter. The class `Timing` must contain—among other—the type of the desired timer and other parameters required by the call-back function of the certain timer, for example if a message has to be resent after a while then the timer must contain the address of the recipient. When the function `insertTiming` of CommC is called then it inserts the received `Timing` in the timers' list of the class `Timer`. To remove (delete) a timer, the layers have to call the `delTiming` function of the CommC.

Beside continuously looking for new messages, the class `Communication` follows the expiration of the timers as well. Both tasks are done using the C++ function `select`. The function is always called with the expiration of the proximate timing. After calling the function, it waits until the time given as parameter expires or a new message arrives on an UDP socket. Unfortunately the function `select` can not detect arrival of new IPC messages, so I made an own function (`isnewIPCmessage`) that checks whether there are new IPC messages in the queues. This function is called at least once every 10 milliseconds by using special timers (*IPC-timer*). If there were no other timers with shorter expiration then this timer always expires after this given time. If another timer expires earlier, then—after the call-back function of the expired timer has returned—my function looks for new messages and resets the IPC-timer. In case of detected new IPC messages the `receiveNewMessage` functions of the appropriate layers are called.

When a timer expires then the CommC removes the given timer from `Timer`'s list of timings and calls the call-back function that is associated to the given *timingType*. The parameters of the given call-back functions are in the fields of `Timing`. The fields were filled out when the class was initiated.

4.3 Class AdaptationLayer

The class `AdaptationLayer` (Figure 4.4) realises the functions of the Blown-up Micronet Protocol's adaptation layer. It is derived from the class `BaseLayer` that defines the virtual function `receiveMessage`. At present the layer adapts the BUMP to UDP/IP protocol.

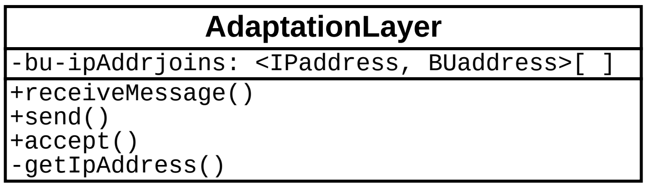


Figure 4.4: The class AdaptationLayer

When initiating the class `Adaptation` then it first fills out an address-translation registry (*bu-ipAddrjoins*) based on a configuration file. This registry contains <Blown-up address, IP address> pairs and is used to determine the IP address of a device that is identified by its Blown-up address (an array of six bytes, i.e. in the implementation an array of six `unsigned char`s) in the Blown-up system. The IP addresses of devices are required for the inter Blown-up system communication; the Adaptation Layer must use these addresses to send messages to remote devices. As the second step in the initialisation the class binds itself to an UDP port on which it will receive messages from other BUMPs in the future. As described in the previous section, the socket identifier will be registered by the class `Communication`. The used portnumber must be given when starting the system.

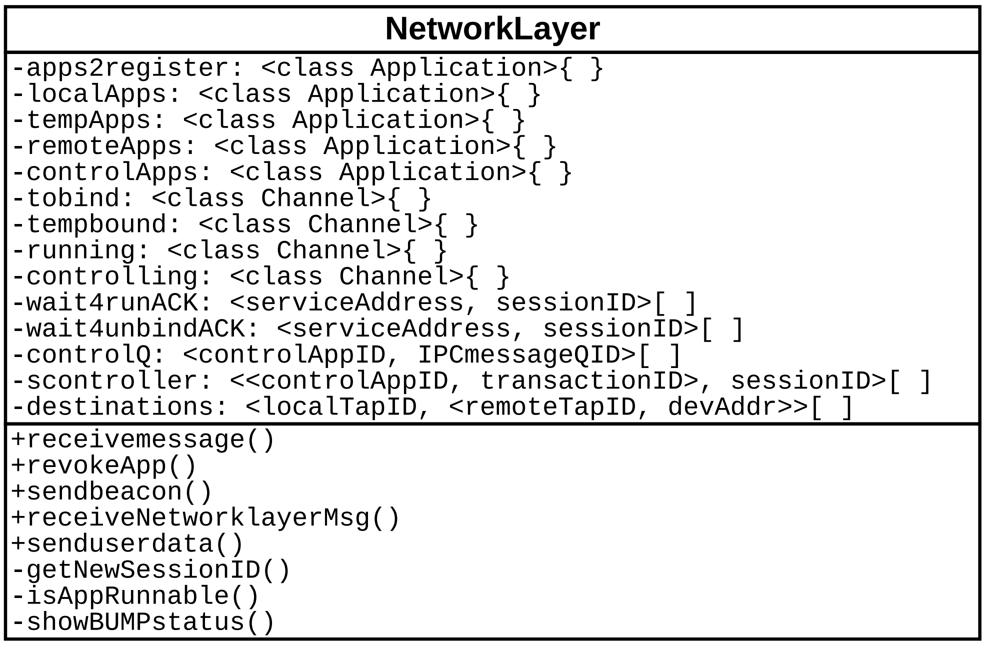
When a new UDP packet arrives to the UDP port of the device then the previously presented class `Communication` detects it and calls the `receiveMessage` function of the class `Adaptation` with the identifier of the socket as parameter. This function will call the function `accept` that receives the message (with the C++ function `recv`) and forwards the embedded message to the BUMP-Network Layer, i.e. it calls the function `receiveNetworklayerMsg`[2] of the class `NetworkLayer`.

The class `Network` that realises the BUMP-Network Layer, can send messages to remote Blown-up systems calling the `send` function of this class. When invoking it then the function first looks into the *bu-ipAddrjoins* address-translation registry and selects the IP address that is associated with the Blown-up address given as parameter to the function. This operation is done by the function `getIpAddress`. Then it puts the received message in a *data* message and sends it (using the C++ function `sendto`) to the device identified by the selected IP address.

[2]more about this function can be found in Section 4.4

4.4 Class NetworkLayer

The class `NetworkLayer` (Figure 4.5) does the tasks of the BUMP-Network Layer and the tasks of the BUMP controller. It is the most complex class in the system. The attributes and functions of the class `NetworkLayer` can be best presented in the course of looking on the main tasks of the Network Layer and BUMPC.



```
NetworkLayer
-apps2register: <class Application>{ }
-localApps: <class Application>{ }
-tempApps: <class Application>{ }
-remoteApps: <class Application>{ }
-controlApps: <class Application>{ }
-tobind: <class Channel>{ }
-tempbound: <class Channel>{ }
-running: <class Channel>{ }
-controlling: <class Channel>{ }
-wait4runACK: <serviceAddress, sessionID>[ ]
-wait4unbindACK: <serviceAddress, sessionID>[ ]
-controlQ: <controlAppID, IPCmessageQID>[ ]
-scontroller: <<controlAppID, transactionID>, sessionID>[ ]
-destinations: <localTapID, <remoteTapID, devAddr>>[ ]
+receivemessage()
+revokeApp()
+sendbeacon()
+receiveNetworklayerMsg()
+senduserdata()
-getNewSessionID()
-isAppRunnable()
-showBUMPstatus()
```

Figure 4.5: The class NetworkLayer (only its main functions are represented)

4.4.1 Registering services and control applications

As described in Section 3.5, the BUMPC has to register the services offered by local and remote applications. To this purpose the class `NetworkLayer` has two lists of services: the list *localApps* contains local and the list *remoteApps* contains remote services and their parameters. Each application is stored in a class `Application` (Figure 4.6) and each of their pins in a class `Pin` (Figure 4.7). The attributes of the classes accord to the attributes described in Section 3.5.2, however the class `Pin` has an additional parameter: *FreeCapacity* indicates the number of channels that can be additional connected to the pin, i.e. actually $MaxCapacity - FreeCapacity$ channels are bound to the pin.

When an application wants to be registered it calls the suitable function of the API that

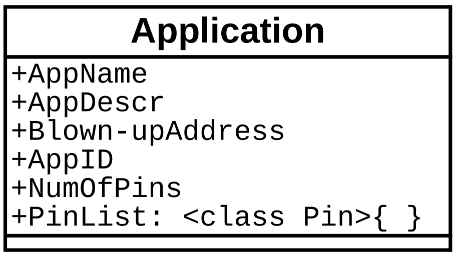


Figure 4.6: The class Application

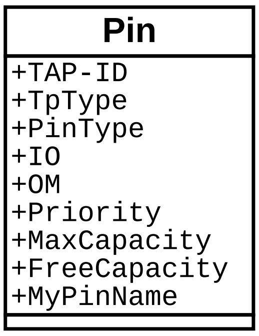


Figure 4.7: The class Pin

first sends a *Hello* message then one or more *RegPinReq* messages and at last a *RegFin* message. When the class `Network` receives the first message then it creates a new subdirectory under `bump`; the name of the subdirectory will be the internal identifier of the application (*AppID*). In addition the class creates an entry in a special list of applications (*apps2register*) that is used to store applications that are just registering themselves. This special list of applications is like the *localApps* and *remoteApps*, however the entries of applications in it do not contain all the pins of the applications, only the pins that were already registered. The entries are continuously updated, always when the class receives a new *RegPinReq* message. In such a case the class first requests a TAP from the BUMP-Transport Layer (i.e. from the class `TransportLayer`) for the pin by calling its `registerPin` function. Then it looks in the list *apps2register* for the application the pin belongs, and appends the pin with its attributes to the pins' list of the selected application. When a *RegFin* message arrives and all pins of the application indicated in the message were successfully registered, then the entry will be moved from the list *apps2register* to the list *localApps* and a positive acknowledgment is sent to the application. Otherwise the entry will be deleted and a negative acknowledgment is returned to the application. Of course during the registration acknowledgments were sent in answer to the *Hello* message and to each *RegPinReq* message too.

Applications registered in *localApps* are continuously advertised by the class `NetworkLayer`: a *beacon* message is periodically broadcasted in the network. For this purpose a timer is set to each application, which always expires after `BEACON_INTERVAL` seconds and then a *beacon* is sent to the other devices in the network. The timer is an entity of the already presented class `Timing` that contains the internal identifier of an application. When the class `NetworkLayer` needs a new timer that can be associated to the beaconing of a newly registered application then it calls the function `insertTiming` of the class `Communication`. From this time on when such a timer expires then the class `Communication` will call the function `sendBeacon` of the class `NetworkLayer`, which will send the certain beacon to the other BU systems in the network.

When a class receives a new beacon (i.e. the list *remoteApps* does not contain the entry of the given service), it puts a new entry in the list *tempApps* that's role is the same as the *apps2register*'s. Then the class requests the specification of the application's pins (i.e. it sends an *appPinsReq* message) and updates the application's list of pins based on the answers until the last *appPinsResp* message arrives from the remote device. When all the pins of the application are known then the class moves the record of the application from the list *tempApps* to the list *remoteApps*.

The registration of a control application is relatively simple: the control application has only to call the API-function `bump_RegisterControlEntity` that sends a *ControlHello* IPC message to the BU system. When the class `NetworkLayer` receives this message, it creates a new subdirectory under `bump` that represents the control application in the filesystem. Furthermore it adds a new entry to the list *controlApps* with the properties of the control application. The list *controlApps* is similar to the lists *localApps* and *remoteApps*, however it contains entries about local control applications. In addition to this two actions, at the same time the class adds a new entry about the control application and its control queue identifier to an associative array called *controlQs* that contains <*controlAppID*, *IPCmessageQID*> pairs. When an IPC message must be sent to a control application then the IPC message queue to use is determined based on this associative array . I have created this special associative array to speed up the selection of the control queue, not to waste time by doing a linear search in the list *controlApps* that could additional contain the identifiers of the control applications queues'.

4.4.2 Creating a new session

The class `NetworkLayer` starts to create a new session when it receives a *TStart* message from a control application. As the first step in the creation of the new session the class generates a new session identifier (*sessionID*) by calling the function `getNewSessionID`, which identifier will be unique in the local BU system, and complemented with the address of the device unique in the network. For the purpose to store *sessionID*s and connections between a *sessionID* and a control application's session (denoted by the identifier of the application and its own *transactionID*) I have created an associative array called *scontroller* that contains <<*controlAppID*, *transactionID*>, *sessionID*> pairs.

The establishing of the session continues with the receiving of one ore more *ConnPins* messages that indicate the two TAPs to bind, and then with the receiving of a *TEnd* message. The handling of these messages is described in details in Section 3.5.3, thus in this section I only present the structures and main functions, which are used on local and remote devices to register these requests and the established sessions.

When a BU system receives a *ConnPins* message, the class `NetworkLayer` must register the request with its status that denote whether the channel could be created so far (i.e. on both endpoints of the channel the pins had free capacity for the new channel). For this purpose I have created a class (class `Channel`, Figure 4.8) that registers one channel with its status, and a list of channels (*tobind*) that contains the previously mentioned class entities. The status of a new entry is always *"waiting for acknowledgment"* until a positive or negative acknowledgment arrives, then the content of the acknowledgment message (*"failure"* or *"success"*) will be placed in the status field.

After receiving a local command to bind a channel to a pin and after receiving a *bind* message from a remote device, the class `NetworkLayer` has to register this request. For this purpose I have created a list of class `Channels` (*tempbound*), wherein the status of an entry indicates whether the channel could be bound to both given endpoints. Since a channel is registered on both devices of the endpoints, at first the status is always *"waiting*

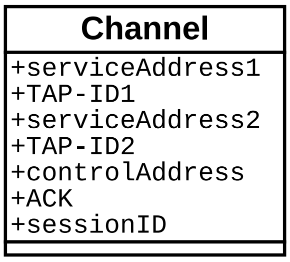


Figure 4.8: The class Channel

for acknowledgment" on devices thats given pin has free capacity for at least one new channel and must forward the *bind* message to a remote device. On these devices the status of this entry is changed when a positive or negative acknowledgment arrives. On devices that do not need to forward the *bind* message and in addition its given pin has free capacity, the status of the new entry will be *"channel bound to the pin successfully"*, however if the pin has no more capacity to bind a channel to itself then the status of the channel will be *"channel could not bound to the pin"*. In addition, if the connection request arrived in form of a *bind* message (i.e. the request was not local) then an acknowledgment is sent back to the sender of the *bind* message indicating the success or failure. The receiver of this acknowledgment message updates its own entry about the given channel based on the content of this message: the new status of the channel's entry will be the status that the entry has about this channel on the other device. Of course all the BU systems, which received a local connection request or a remote request in form of a *bind* message that they needed to forward, send an acknowledgment about the result of the channel's establishment to the sender of the request. The status information of the entries in *tobind* are updated based on these acknowledgments.

When the class `NetworkLayer` receives a *TEnd* message, then it looks into its list *tobind* and checks whether a positive acknowledgment has arrived for each connection request. If not, then it tears down the establishment of the session, otherwise it continues the procedure by moving all the entries of the list *apps2register* that belongs to the given session to the list *controlling*. The list *controlling* is like the list *tobind*, however contains channels that were successfully initialised by a local control application. In addition to the movement of channel entries, the class sends a broadcasted *run* message to the other devices, sets a timer for the acknowledgment of the *run* message, and at the same time it puts the identifiers and addresses of the services that takes part of the session into the list *wait4runACK*. This list is used to register the applications that must send an acknowledgment to the *run* message. When an acknowledgment arrives from a remote application then the entry of the application will be deleted. However when the timer of the acknowledgment expires and there is at least one entry about an application in the list *wait4runACK*, which should have sent an acknowledgment then the *run* message will be resent.

The BU systems that receive a *run* message have to check if they have entries in their list *tempbound* that belong to the session indicated in the body of the received message. If yes, then they have to check for each application that has at least one pin that takes part of the indicated session whether all the mandatory pins of the application are used by the session. To this purpose I have created a function called `isAppRunnable`: it returns "yes" if the service given as parameter is ready to operate, otherwise no. When this function finds out that a service is ready to start, then all the entries in the list *tempbound* that belongs to this certain application and session will be moved to the list *running*. The list *running* contains the channels that are part of a "running" session, and is similar to list *tempbound* in structure, however the status field of an entry indicates whether the focus of the pin the channel is connected to is on the certain channel or not. In addition to the movement of the entries an acknowledgment is sent back to the device of the session's creator.

If a newly established channel is connected to a pin, which has no other channels then the channel must become the focus. The focus will be registered in two forms: on the one hand the status field of the channel's entry in the list *running* will be set to *"has focus"*, on the other hand a new entry will be put in the associative array *destinations*. The data packets sent by the users are forwarded based on this associative array; each of its entries contains a local *TAP-ID* and the remote endpoint of the active channel (remote *TAP-ID* and device address of the remote TAP). Like the associative array *controlQs*, I have created the associative array *destinations* specially to speed the selection of a pin's channel. Such a way the destination of a packet is determined quite quickly, it needs surely less time than a linear search in the list *running*. In addition to the previous actions if an output pin is connected to the local TAP that had no channels connected to itself just now then the class `NetworkLayer` must call the function `unlockOutputPin` of the class `AdaptationLayer` that unlocks the given pin (in the future data can be sent through the pin), otherwise the remote pin must be unlocked in the same way.

4.4.3 Sending data

The class `TransportLayer` sends data from a TAP to another TAP by calling the function `sendUserData` of the class `NetworkLayer`. The parameter *senderTapID* of the function indicates the source of the data: the destination of the data packet will be determined based on this parameter. When receiving a data to forward then the function `sendUserData` looks into the associative array *destinations* and searches for the entry associated to the *senderTapID*. The returned entry contains the device address and the TAP's identifier (*TapID*) of the channel's remote endpoint that must receive the data packet. The data is forwarded to the device of this TAP by calling the function `send` of the class `AdaptationLayer`.

When a data packet arrives from the class `AdaptationLayer` to the class `NetworkLayer` then this latter class must forward the received data to the class `TransportLayer` by calling its function `receiveUserData`.

4.4.4 Tearing down a session

In case of tearing down a session all the entries in the BU systems about the certain session must be removed in a specified order. However before removing any entries of any registers the BU system that has created the session must put the identifiers of the applications that are part of the session—complemented with their device address—into the list *wait4unbindACK*. This list is used to check whether all the acknowledgments has been arrived for an *unbind* message. When an acknowledgment arrives then its entry will be removed from the list, however when the timer of a certain *unbind* message expires and the list *wait4unbindACK* hast at least one entry about the certain session then the *unbind* message will be resent.

When a device receives an *unbind* message then first the focus must be changed on pins thats focus is on a channel that belongs to the certain session. If only one channel is connected to the certain pin then the focus can not be changed, but this is not a problem. In the next step each entry must be removed from the associative array *destinations*, which belongs to the certain session, and at the same time each pin thats entry was removed from this associative array must be locked by calling the function `lockOutputPin` of the class `TransportLayer`. Consequently all the output pins that had more than one channel connected to themselves can send data through one of their additional—existing—channel, and the output pins that had only the removed channel connected to themselves stop to send data. After the just now described actions, the entries about the certain session in lists *tempbound* and *running* must also be removed. At the same time if an input pin has its focus on a channel that must be removed then its focus must be changed. Similar to the focus changing action on an output pin when removing a channel, it is no problem if the focus can not be changed. In this case there will no channel bound to the certain pin.

The BU system that has sent an *unbind* message continuously checks for new acknowledgments until all of them has been arrived. When there are no more acknowledgments to receive then the class `NetworkLayer` has to remove all its entries about the certain session from the lists *tempbound* and *controlling*. If a session was ”running” then the list *controlling* had entries about the certain session, however if the session was not ”running”, only under establishment then the list *tempbound* had correspondent entries. As the last step in the process of tearing down a session the entry about the session in the associative array *scontroller* must be removed. After all this actions the session is torn down, the BU systems in the PAN have no more entries in their registries about the certain deleted session.

4.4.5 Unregistering services and control applications

A service will be unregistered when the class `NetworkLayer` receives a *RevokeApp* message from an application. In such a case the class tears down all the sessions wherein the service took part and then calls the function `unregisterPin` of the BUMP-Transport Layer for each pin of the certain application. The called function unregisters all the pins of the given application one by one. In addition the class removes the entry of the application from the list *localApps* and calls upon the class `Communication` to delete the timer that was responsible for the periodically beaconing of the certain application. This latter is done by calling the function `delTiming` of the class `Communication`.

The BU systems in the PAN will be informed of the service's revoking by the lack of the service's beacons. In such a case every BU system has to remove all its entry about the certain application: it has to delete the corresponding entries from the lists *remoteApps* and *tempApps*.

The unregistration of a control application is the effect of a *ControlBye* message sent by a control application. In such a case all the sessions must be torn down that have been created by the certain control application. After this, two entries must be removed: the entry about the control application in the list *controlApps* and the entry about the IPC message queue of the control application from the associative array *controlQs*. If both were successfully removed then the control application has been unregistered.

4.5 Class TransportLayer

The class `TransportLayer` (Figure 4.9) realises the tasks of the BUMP-Transport layer. I have implemented a base transport module that has no flow control and do not use acknowledgments. When it receives data from a user application through a TAP then it immediately forwards it to the TAP on the other endpoint of the connected channel.

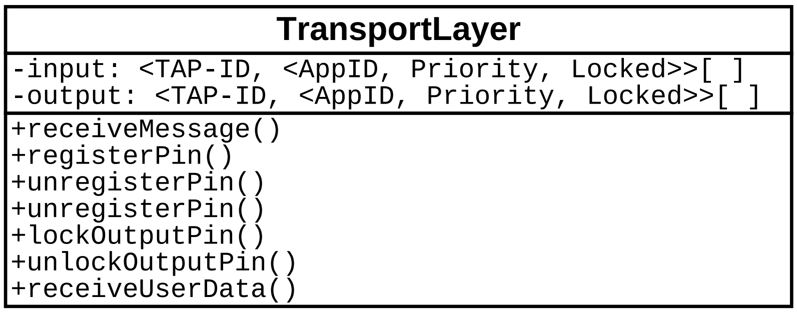


Figure 4.9: The class TransportLayer

When a new applications starts then it has to register itself by the BU system. As part of this registration the application receives a TAP to each of its pin. A TAP is associated to a certain pin by this class when calling its function `registerPin`. This function first creates a file based on the name of the pin in the subdirectory of the application, and then it generates a unique IPC message queue identifier (*TapID*) based on the filename of the pin using the C++ functions `ftok` and `msgget`. If a user application wants to send data through one of its pin to another pin or wants to receive data through one of its pins from another pin then it can forward the data to the BU system or receive data from the BU system by using the message queue identified by this just now generated identifier. After the generation of the identifier the class `TransportLayer` must save the identifier of the TAP and in addition some properties of the corresponding pin. This properties are:

- the identifier of the application the pin belongs to (*AppID*),
- the priority of the pin (*Priority*), and
- a boolean variable (*Locked*) whether the TAP associated to the pin is locked or not, that is whether data can be sent through the pin or not.

If the pin associated to the TAP to register is an input pin then the above presented record must be put in the associative array *input*, otherwise—if the pin is an output pin—then in the associative array *output*. The records of these associative arrays will be used when a pin sends data to the BU system: for example data will only be sent to the remote endpoint of a channel if the local TAP is unlocked (the variable *Locked* of the certain TAP's record is set to ”nonlocked”), i.e. the pin connected to the remote endpoint of the channel has its focus on the certain channel.

In addition to the creation of a file based on the name of the pin and to the registration of the TAP, the function `registerPin` has to call the function `insertmsgqID` of the class `Communication`. By calling this function with *TapID* as parameter the class `TransportLayer` indicates to the class `Communication` that there is a new IPC message queue that must be checked periodically for new messages from now on. After this last action the function `registerPin` gives back to the caller whether it successfully registered the pin given as parameter or not.

By revoking an application the pins of the certain service must be unregistered. In such a case the class `NetworkLayer` calls the function `unregisterPin` of the class `TransportLayer`. This function removes the entry about the certain pin from the associative array *input* or *output*, and in addition deletes the file from the filesystem, which belongs to the pin that must be pin unregistered. Thus the TAP the pin was connected to is free again and can be associated to a new pin.

To support changing focus on the pins there are two functions in the class `TransportLayer`: `lockOutputPin` and `unlockOutputPin`. The former one changes the attribute *Locked* of a certain TAP to ”locked” that means data can not be sent through the pin, the latter one does the negative of the former action: it sets the attribute *Locked* to ”unlocked” and so data can flow through the pin from this time on. Both functions can be called by the class `NetworkLayer`.

The main task of the class `TransportLayer`, the forwarding of the data from a pin to another pin is quite simple. When a service sends data through one of its pins then the data arrives to the `TransportLayer` in form of an IPC message. The arrival of such a message is detected by the class `Communication`, since all the TAPs that are in use have been registered by this class. Thus when new data arrives then the class `Communication` calls the function `receiveMessage` of the class `TransportLayer` and gives it as parameter the identifier of the IPC message queue that contains the new message. After this the function `receiveMessage` has to check if the pin is locked or not, and if not then it has to forward the message retrieved from the certain IPC message queue to the class `NetworkLayer` by calling the function `sendUserData` of this latter class. Otherwise, if the pin is locked then the data is thrown away, since there are no channels connected to the TAP, or the pin at the remote endpoint of the channel has its own focus on an other channel.

Data arrives from a channel to a local TAP when the class `NetworkLayer` calls the function `receiveUserData` of the class `TransportLayer`. In such a case the preceding function has only to forward the data to the adequate pin through the message queue identified by the message queue identifier in the header of the received message.

4.6 The library of the application programming interface

As described by the presentation of the BUMP, the control and user applications can access the BU system through an API. Thus I have created a library that collects the functions defined by the BUMP-API in Section 3.6. This library can be used by the developers who want to implement services for BU systems.

The majority of the functions' task is quite simple: they send adequate messages to the

BUMPC or to the class `TransportLayer` and receives adequate messages from the BUMPC or from the class `TransportLayer`. The communication follows with the BUMPC via the BUMPC's IPC message queue thats identifier is determined based on the directory `bump` using the C++ functions `ftok` and `msgget`, and via the control queues of the applications (both user and control) thats identifier is generated based on the name of the applications' directories. Similar to this, the TAPs are accessed through the IPC message queues thats identifier is determined based on the representation of the pins in the filesystem. The messages are C++ structures that contain a message identifier, are defined in a header file and contain the parameters passed to the functions.

In the rest of this section I will present the API functions that have additional tasks as to only send or receive an IPC message. These functions and their tasks are the following:

- `bump_GetServicesReset`: in addition to sending an *AppListReq* message to the BUMPC, the function removes all the expired *AppItems* messages—that were sent in answer of the previous similar request—from its own control queue. Furthermore—as the effect of this functions' call—the message queue will contain all the new *AppItems* messages that were sent by the BUMPC in answer of the previous request. After this call the control application can get to know all the available services in the PAN by calling the API function `bump_GetApplications` several times, until it returns 0. This return value indicates that the application received the last entry of the available service's list, that is there are no more *AppItems* messages in the IPC message queue.
- `bump_PinsReset`: this API function is similar to the previous presented function, however it removes the *PinItems* messages from the control queue instead of the *AppItems* messages. Furthermore in the effect of this call the list will contain the new *PinItems* messages that can be queried using the function `bump_GetPins`.
- `bump_CreateSessionStart`: since a control application refers to each of its own sessions by using a unique transaction identifier (*TransactionID*), so before sending a *TStart* message to the BUMPC the API has first to provide such a new identifier. This is done by calling the function `getNewTransactionID` that generates a new, till now not used identifier (using the C++ function `rand`). This identi-

fier will be registered in the array *transactions* no to generate this identifier a second time, furthermore the identifier will be returned to the caller of the function `bump_CreateSessionStart`. From this time on the creator of the certain session must use this identifier to refer to its own session.

- `bump_EndSession`: after tearing down a session, the identifier of the transaction (*TransactionID*) must be removed from the array *transactions*. This is done by this function after it sent the *SessionEnd* message to the BUMPC.

4.7 The test of the implementation

To make sure of the BU system's correct functioning I have tested it continuously during the implementation, and also after it. I had to test whether all the messages caused the right action, whether all the lists and associative arrays contained always the necessary entries, and whether all the messages were resent, which acknowledgment was missing.

As the first step by the testing I modified the BU system to log every messages with its parameters, which was sent or received. Thus I could check whether all the messages was sent and received in the right order as it is required by the protocol. Then I have implemented some simple applications that registered certain pins by the BU system, and a control application too that could automatically bind and unbind the pins of the previous applications in form of a session. However before testing the creation of a new session I had to test whether the services are well registered an advertised to the other BU systems. So I have started several applications on different devices and have checked whether the available services appears on the remote devices, and whether they disappear when the service is revoked or when the device that offers the service stops beaconing (i.e. the provider of the service moved away and so the two devices are too far to communicate with each other).

When all the services were successfully registered and advertised by the BU system then I have started to connect this services together in form of a session and then to tear this sessions down. By testing this task the flow of the messages gave not enough information about the right functioning of the BU system, so I have defined a new control message that can be sent from a control application to the BU system. I have also defined an API function to be able to send such a message. As the effect of this message the BU system visualizes all of its entries registered in different lists and associative arrays (it can best compared to a *dump*). Thus, creating a special control application that calls this previous function, I was able to check in every moment whether all the entries of the BU system are right (no entry is missing, no is unnecessary, and no entry has wrong attributes).

After I have determined that services could be successfully registered and advertised, furthermore the services could be bound together with no problem, then I have started to test the reactions of the BU system if an UDP message vanishes. In such a case the system must try to resend this lost message, however stop the retransmission after several times. To this purpose I have modified the successfully functioning BU system. I have checked what happens if a certain message will be not sent that had to, or what happens if a certain message is sent only at the second or third—repeated—request. I have found that the system worked well: all the request that must be repeated in case of a failure were successfully sent a second or third time.

When I have noticed that the system can create and tear down the sessions successfully I had to test the sending of data. To this purpose I have implemented several user applications that were able to send data through their output pins and to receive data through their input pins. I have determined that the data sent by the sender successfully arrived to the receiver.

To summarize the foregoings, after the tests I assert that my system works well—as specified by the BUMP in Chapter 3—, only the function of focus changing is missing due some failures in the design of the BUMP. I will present this latter problem in Section 4.8.

In addition to the test methods described previously—after that I have finished the implementation and testing of my BU system—I could test the system with a control application that had a graphical user interface (GUI) and test it with some more complex user applications. My colleague created for the BU system a file server and client, an mp3 player front-end and mp3 player, furthermore a control application with a GUI [Kovacs03], so I had the opportunity to test my system with this real applications too. The applications have used my BU system without any problem.

4.8 Evaluation of the Blown-up Micronet Protocol

After the implementation of the BUMP I state that the protocol realises the Blown-up concept, however I found two errors in the protocol. The first one is not a significant error: when a device receives an *unbind* message and returns an acknowledgment in answer of this previous message to the source of the request, furthermore this *unbindACK* is lost, then the source will never receive an acknowledgment for its *unbind* message. As described in Section 3.5.4, a device answers to an *unbind* message only if it takes part of the session, i.e. it has several entries about the session in its registries. However in such a case the device also immediately deletes all its channel entries about the certain session. Thus if a device that was part of the deleted session receives a second or third *unbind*, it will not send any acknowledgments to the source again, since it has no more entries in its registries about the certain session. This problem could be solved if the entries about a session would—instead of immediately deleting—first be moved to a temporary register, and be only after a certain time deleted. The solution at present causes sometimes unnecessary *unbind* messages, however the sessions will always be torn down.

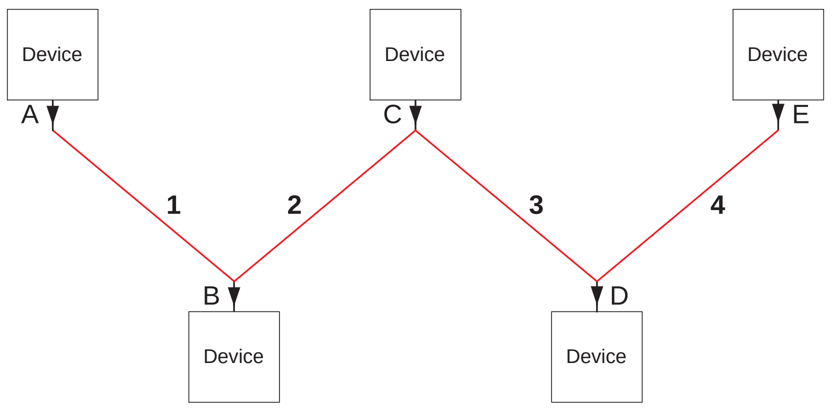


Figure 4.10: Example for the occurrence of a focus changing error

The second error of the protocol is much more important. The problem is now about the focus changing function. On Figure 4.10 a case can be seen, where there are five pins: two input pins (B, D) and three output pins (A, C, E) where the pin B is connected with pins A and C, and pin D with pins C and E. Let the focus be by pin B on the channel 1, by pin C on the channel 3, and by pin D on channel 4. Now, when the device that has the pin B switches the focus on its pin from channel 1 to channel 2 then it has to send an *unlockOutputPin* message to the device that has the pin C. So the pin C will be unlocked, but this is an error, since its focus is on the channel 3 and at this time it should not send any data to pin D. However the pin C will send data to the pin D that will think it has received data from pin E. Consequently—in addition to the pins—the channels should be locked too, that is a channel should be locked if the focus of the input pin connected to either endpoints of the channel is not on the certain channel. Therefore a pin should be locked if there are no channels bound to it, or when the pin is an output pin and the focus of the input pin on the other end of the channel has its focus on another channel. Since the focus changing function has this error, I have not implemented it in my system.

Though the protocol enables the easily binding of services in a PAN together, I have several ideas, which could make the BU systems more useful. One of this ideas is the dynamically rebinding of the pins, that is a task in the BU system would be able to follow the ”mobile” user. For example, if the user listens to music and moves from a room to another, then the music would follow her or him, that is first a speaker in the first room would be used, then another one in the second room without restarting the actually played song. At present the BU system can solve this problem only by stopping playing the music and then restarting it.

Another function that could be added to the BU system in the future is the collective changing of focus, i.e. a focus would be changed only if all the other focus could be changed that were required by the same request. Thus the changing of a focus would be a two-phase procedure, like the creating of a session by the actual protocol.

The BUMP could be in addition expanded by supporting more than one control application per device at the same time. Although at present the BUMP do not restrict the number of simultaneously running control entities per device, but the disassociation of more control applications is not fully defined in terms of security. The actually protocol enables to a control application to instruct the BUMPC in the name of another control application, since a control application can put the identifier of another control entity in the control message without any consequences. Thus it would be necessary to define priorities and such a way support the parallel running of several control applications.

It would be perhaps worth to consider whether to deny the applications to send or receive data if the focus of their pins are on channels that belongs to different sessions. At present all the sessions can be joined by binding channels of different sessions to the same pin, but in the future it could be a special call for this purpose and thus data could not get out of a session or of a sessions' group. With this feature the data sent through the BU system would be protected against unauthorized applications.

Chapter 5

Conclusion

Nowadays—as the computer technologies continuously develops—we are able to establish intelligent environments that consist of connected intelligent devices. In such intelligent environments the devices can effectively cooperate with each other, can distribute the tasks to special devices, furthermore can perceive the intentions of the users. The devices seamlessly integrates in our life and becomes a natural part of it. The actually used approach on computers can be forgotten related to most of the tasks in this concept. In the intelligent environment people use only groups of connected special services to execute their tasks. For example a user can write a composition by connecting a display, a keyboard and a pointer device to a special text editor device.

In my Thesis I have presented the Blown-up concept that was designed to connect devices in a Personal Area Network forming an intelligent environment. I have shown that the Blown-up system hides the distributedness of the devices, and so of the services: the services see the others as they were located on the same devices. The distributed behavior of services appears only by the control applications that can connect the services together. If there are some sensor devices in the intelligent environment and special control devices have some knowledge about the users, about their task, moreover about the setup (physical and logical) of the environment, then the binding procedure can be done automatically. Otherwise the user must connect the services using the graphical user interface of a control application.

As part of my thesis I have implemented a system that realise the BUMP. At the moment the system contains one base transport module in the BUMP-Transport layer and one adaptation module in the BUMP-Adaptation layer, which adapts the BUMP to IP based networks. I have also created some test applications to check whether my system works well. After several tests I have stated that my system works without errors and realises the main tasks of the BU concept according to the specification. The system can successfully register applications and advertise the services provided by the devices, it is able to create groups of connections between pins of the services, furthermore data can be sent on the established channels without problem. I have also created a library that contains the API-calls defined for the BUMP. By the help of this library the developer can easily implement applications designed for BU systems.

In the future—by examining the implemented system—the BUMP can be improved and the features of the system can be extended. During the implementation I have also studied the BUMP and have pointed out its weaknesses, furthermore I have given propositions to solve them. I have found a situation where *unbind* messages have to be sent, but unnecessarily. Then I have detected that the specification of changing focus has a significant error: it is not enough to lock an output pin when the input pin on the other end of the channel lost the focus, the channel should be locked too. In addition to detecting the weaknesses of the protocol I have proposed some features that could be added to the BUMP. Among others I suggested the dynamically modifying of sessions and the collective changing of focus.

In the next time abundance of services have to be created for the system, and based on their claims new features must be added to the concept.

Appendix A

Messages used in BUMP

Source	Destination	Message (parameters)
UA	BUMPC	Hello(AppID, AppName, AppDescr)
UA	BUMPC	RegPinReq(AppID, TpType, PinType, I/O, M/O, Priority, Capacity, MyPinName)
UA	BUMPC	RegFin(AppID)
UA	BUMPC	RevokeApp(AppID)
UA	BUMPC	UnRegPinReq(AppID, MyPinName)
CA	BUMPC	ControlHello(controlAppID)
CA	BUMPC	ControlBye(controlAppID)
CA	BUMPC	AppListReq(controlAppID)
CA	BUMPC	GetInfo(serviceAddr, serviceAppID, controlAppID)
CA	BUMPC	TStart(controlAppID, transactionID, NumOfPairs)
CA	BUMPC	ConnPins(linkAddr1, AppID1, MyPinName1, linkAddr2, AppID2, MyPinName2, transactionID)
CA	BUMPC	TEnd(transactionID)
CA	BUMPC	PinListReq(controlAppID, serviceAddr, serviceAppID)
CA	BUMPC	SessionEnd(controlAppID, transactionID)
CA	BUMPC	ChangeRemoteFocus(serviceAddr, AppID, MyPinName)
CA	BUMPC	ChangeLocalFocus(AppID, MyPinName)

Continued on next page

Source	**Destination**	**Message (parameters)**
BUMPC	UA	RegPINResp(AppID, PinType, TAP-ID)
BUMPC	CA	AppListResp(serviceAddr, serviceAppID)
BUMPC	CA	Info(serviceAddr, serviceAppID, AppDescr)
BUMPC	CA	PinItems(PinType, TpType, I/O, M/O, Capacity, TAP-ID, MyPinName)
BUMPC	CA	SessionTerminated(bindID)
BUMPC	CA	ConnPinsresp(success, linkAddr1, AppID1, MyPinName1, linkAddr2, AppID2, MyPinName2, transactionID)
BUMPC	CA	TEndResp(success, transactionID)
NL	TL	DATA(TpType, TAP-ID, frameNum, userData)
NL	TL	DATAACK(TpType, TAP-ID, frameNum)
TL	NL	DATA(TpType, TAP-ID, frameNum, userData)
TL	NL	DATAACK(TpType, TAP-ID, frameNum)
NL	NL	data(destAddr, destTAP-ID, senderAddr, userData)
UA	TL	appData(userData)
TL	UA	appData(userData)
BUMPC	BUMPC	beacon(senderAddr, AppID, AppName)
BUMPC	BUMPC	appPinsReq(destAddr, senderAddr, destAppID, queryID)
BUMPC	BUMPC	appsPinsRespStart(destAddr, senderAddr, sourceAppID, AppDescr, NumofPINs)
BUMPC	BUMPC	appsPinsResp(PinType, TpType, I/O, M/O, Capacity, TAP-ID, MyPinName)
BUMPC	BUMPC	bind(linkAddr1, TAP-ID1, linkAddr2, TAP-ID2, senderAddr, controlAddr, sessionID)
BUMPC	BUMPC	bindACK(destAddr, linkAddr1, TAP-ID1, linkAddr2, TAP-ID2, senderAddr, sessionID)
BUMPC	BUMPC	bindNAK(destAddr, linkAddr1, TAP-ID1, linkAddr2, TAP-ID2, senderAddr, sessionID)
BUMPC	BUMPC	run(BROADCAST, controlAddr, sessionID)
BUMPC	BUMPC	runACK(destAddr, senderAddr, sessionID)

Continued on next page

Source	**Destination**	**Message (parameters)**
BUMPC	BUMPC	unbind(BROADCAST, controlAddr, sessionID)
BUMPC	BUMPC	unbindme(controlAddr, senderAddr, senderAppID, sessionID)
BUMPC	BUMPC	unbindACK(controlAddr, senderAddr, senderAppID, sessionID)
BUMPC	BUMPC	lockOutputPin(destaddr, senderAddr, TAP-ID)
BUMPC	BUMPC	unlockOutputPin(destAddr, senderAddr, TAP-ID)
BUMPC	BUMPC	lockOutputPinACK(destAddr, senderAddr, TAP-ID)
BUMPC	BUMPC	unlockOutputPinACK(destAddr, senderAddr, TAP-ID)
BUMPC	BUMPC	chFocus(destAddr, senderAddr, TAP-ID)
BUMPC	BUMPC	chFocusACK(destAddr, senderAddr, TAP-ID)
BUMPC	TL	PinReq(AppID, MyPinName, Priority)
TL	BUMPC	PinResp(AppID, MyPinName, TAP-ID)

Bibliography

[Weiser91] Mark Weiser: *"The Computer for the 21st Century"*, Scientific American, September 1991.

[Weiser93] Mark Weiser: *"Some Computer Science Issues in Ubiquitous Computing"*, Communications of the ACM, July 1993.

[Gilder93] George Gilder: *"Dark Fibre, Dump Network"*, Forbes ASAP, December 1993.

[Tatai97] P. Tatai: *"Open Vocabulery Speech Recognition – Brief State Report on a research Project"*, Proceedings of the Polish-Czeh-Hungarian Workshop on Circuits Theory, Signal Processing and Applications, September 3-7, 1997, Budapest, pp.52-57

[Multivox92] G. Olaszy, G. Gordos and G. Németh, *"The MULTIVOX multilingual text-to-speech converter"*, in: G.Bailly, C. Benoit and T. Sawallis (eds.): Talking machines: Theories, Models and Applications, Elsevier, 1992, pp. 385-411.

[Roska&Chua93] T. Roska and L. O. Chua, *"The CNN Universal Machine: An analogic array computer"*, IEEE Transactions on Circuits and Systems-II, Vol. 40, pp. 163-173, March 1993.

[WLAN99] *"IEEE Std 802.11, 1999 Edition"*, http://standards.ieee.org/catalog/olis/lanman.html

[HLAN2] *HiperLAN2 overview*, http://www.hiperlan2.com/WhyHiperlan2.asp

[JH98] J. Hartsen: *"BLUETOOTH – The universal radio interface for ad hoc, wireless connectivity"*, Ericsson Review No. 3, 1998

[BBSpec] *"Bluetooth Baseband Specification"*, http://www.bluetooth.com

[DPR00] S. Das, C. Perkins, E. Royer: *"Performance Comparison of Two On-demand Ad hoc Routing Algorithms"*, Proceedings of the IEEE Conference on Computer Communication, March 2000.

[PB94] C. Perkins, P. Bhagwat: *"Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers"*, SIGCOMM'94

[PC97] Vincent D. Park and M. Scott Corson: *"A Highly Adaptive Distributed Routing Algorithm for Mobile Wireless Networks"*, Proceedings of IEEE INFOCOM '97, Kobe, Japan, April 1997

[BMJHJ98] J. Broch, D. A. Maltz, D. B. Johnson, Y. Hu, J. Jetcheva: *"A Performance Comparison of Multi-Hop Wireless Ad Hoc Network Routing Protocols"*, MobiCom '98.

[Satya01] M. Satyanarayanan: *"Pervasive Computing: Vision and Challenges"*, IEEE Personal Communications, August 2001.

[McCrory00] A. McCrory: *"Ubiquitous? Pervasive? Sorry, the don't compute*, Computer World, March 2000.

[Aura02a] D. Garlan, D. P. Siewiorek, A. Smailagic, P. Steenkiste: *"Aura: Toward Distraction-Free Pervasive Computing"*, IEEE Pervasive Computing, 2002.

[Aura02b] João Pedro Sousa, David Garlan: *"Aura: an Architectural Framework for User Mobility in Ubiquitous Computing Environments"*, 3rd Working IEEE/IFIP Conference on Software Architecture, Montreal, 2002.

[Aura00] Z. Wang, D.Garlan: *"Task-Driven Computing"*, Carneige Mellon University Technical Report CMU-CS-00-154, http://reports-archive.adm.cs.cmu.edu/cs2000.html, May 2000.

[Oxygen] *„MIT Project Oxygen"*, http://oxygen.lcs.mit.edu

[one.world00] R. Grimm, T. Anderson, B. Bershad, D. Wetherall: *"A system architecture for pervasive computing"*, appeared in the Proceedings of the 9th ACM SIGOPS European Workshop, pages 177-182, Kolding, Denmark, September 2000.

[one.world01] R. Grimm, J. Davis, B. Hendrickson, E. Lemar, A. MacBeth, S. Swanson, T. Anderson, B. Bershad, G. Borriello, S. Gribble, D. Wetherall: *"Systems directions for pervasive computing"*, appeared in the Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII), pages 147-151, Elmau, Germany, May 2001.

[one.world03] R. Grimm, J. Davis, E. Lemar, A. MacBeth, S. Swanson, T. Anderson, B. Bershad, G. Borriello, S. Gribble, D. Wetherall: *"Programming for pervasive computing environments"*, submitted for publication, 2003.

[PortoWeb99] *"Portolano/Workscape: Charting the new territory of invisible computing for knowledge work"*, Online Documentation, http://portolano.cs.washington.edu/proposal/

[Porto99] M. Esler, J. Hightower, T. Anderson, G. Borriello: *"Next Century Challenges: Data-Centric Networking for Invisible Computing: The Portolano Project at the University of Washington"*, Mobicom '99.

[EasyLiving00] B. Brumit, B. Meyers, J. Krumm, A. Kern, S. Shafer: *EasyLiving: Technologies for Intelligent Environments*, Handheld and Ubiquitous Computing, September 2000.

[Kovacs03] B. Kovács: *Design and Implementation of Distributed Applications in Ad Hoc Network Environment*, Master's Thesis, May 2003.

[BlownUp02] G. Biczók, K. Fodor, B. Kovács, Á. Szabó: *Blown-up rendszer tervezése és megvalósítása*, Science Conference for Students, Budapest Univeristy of Technology and Economics, Budapest, October 2002.